



Community Experience Distilled

Jasmine JavaScript Testing

Second Edition

Test your JavaScript applications efficiently using Jasmine and React.js

Paulo Ragonha

[PACKT] open source*
PUBLISHING community experience distilled

www.it-ebooks.info

Jasmine JavaScript Testing

Second Edition

Test your JavaScript applications efficiently
using Jasmine and React.js

Paulo Ragonha

[PACKT] open source 
PUBLISHING community experience distilled

BIRMINGHAM - MUMBAI

Jasmine JavaScript Testing

Second Edition

Copyright © 2015 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

Second edition: April 2015

Production reference: 1210415

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

ISBN 978-1-78528-204-1

www.packtpub.com

Credits

Author

Paulo Ragonha

Project Coordinator

Suzanne Coutinho

Reviewers

Hany A. Elernary

Ryzhikov Maksim

Veer Shubhranshu Shrivastav

Sergey Simonchik

Proofreaders

Paul Hindle

Linda Morris

Indexer

Tejal Soni

Commissioning Editor

Amarabha Banerjee

Production Coordinator

Aparna Bhagat

Acquisition Editor

Larissa Pinto

Cover Work

Aparna Bhagat

Content Development Editor

Manasi Pandire

Technical Editor

Anushree Arun Tendulkar

Copy Editor

Sarang Chari

About the Author

Paulo Ragonha is a software engineer with over 7 years of professional experience. An advocate of the open Web, he is inspired and driven to build compelling experiences on top of this ubiquitous platform.

He loves to hack, so you will often see him wandering around in conferences or attending *hackathons*. His most recent professional experiences ranged from DevOps (with Chef and Docker) to moving up the stack with Node.js, Ruby, and Python and all the way toward building single-page applications (mostly with Backbone.js and "ad hoc" solutions).

Passionate about automation, he sees testing as a liberating tool to enjoy the craft of writing code even more. Back in 2013, he wrote the first edition of the book *Jasmine JavaScript Testing*, Packt Publishing.

Paulo has an amazing wife, who he loves very much. He lives in beautiful Florianópolis, a coastal city in the south of Brazil. He is a casual speaker, a biker, a runner, and a hobbyist photographer.

About the Reviewers

Hany A. Elemary is a software engineer / technical team lead at OCLC in Columbus, Ohio, currently working on the next generation of mobile/web apps (<http://www.worldcat.org> and WorldCat for local institutions). He has been blessed with diverse experience while working for multiple companies (from small software shops to large corporations) and seeing different releasable software strategies. He has a clear focus and passion for mobile/web UI design, interactions, usability, and accessibility. When there is time, he enjoys playing his acoustic guitar, AnnaMaria.

Special thanks to my close friends and family for always pushing me to be better in every aspect of life.

Ryzhikov Maksim is a 27-year-old software developer from Saint Petersburg, Russia.

He develops complex web applications. He graduated from the physics faculty at Saint Petersburg State University. His journey into the world of software development started not so long ago—5 years ago.

His brother invited him to join the team that developed programs for American hospitals, as an HTML developer.

Ryzhikov started with developing a simple, static site for hospitals and then studied JavaScript, Ruby, and SQL and worked as a full-stack developer. In 5 years of work in the area of IT, he has worked in various projects and teams. He developed medical systems, dating sites, web mail (Yandex.Mail), and now he helps develop tools for developers at JetBrains.

Veer Shubhramshu Shrivastav is an Indian software developer working with Tata Consultancy Services since 2013 and is a former research intern at IIIT-Delhi. He has worked on different technologies, such as PHP, Moodle, jQuery, AngularJS, RequireJS, Android, Jasmine, Ionic, and so on, and also takes an interest in cryptography, network security, and database technologies. He has worked with various Indian IT start-ups, helping them as a software architect.

With his interest in the open source community, he developed a Pro*C library named CODBC, which is available at <http://codbc.com>. The library enables an object-oriented approach to connect C++ and Oracle Database.

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 the 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

For support files and downloads related to your book, please visit www.PacktPub.com.

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.



<https://www2.packtpub.com/books/subscription/packtlib>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read 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 a 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 9 entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface	v
Chapter 1: Getting Started with Jasmine	1
JavaScript – the bad parts	1
Jasmine and behavior-driven development	3
Downloading Jasmine	4
Summary	6
Chapter 2: Your First Spec	7
The Investment Tracker application	7
Jasmine basics and thinking in BDD	8
Setup and teardown	14
Nested describes	18
Setup and teardown	19
Coding a spec with shared behavior	19
Understanding matchers	20
Custom matchers	21
Built-in matchers	26
Summary	32
Chapter 3: Testing Frontend Code	33
Thinking in terms of components (Views)	34
The module pattern	35
Using HTML fixtures	36
Basic View coding rules	40
The View should encapsulate a DOM element	41
Integrating Views with observers	43
Testing Views with jQuery matchers	48
The toBeMatchedBy jQuery matcher	49
The toContainHtml jQuery matcher	50
The toContainElement jQuery matcher	50

The toHaveValue jQuery matcher	50
The toHaveAttr jQuery matcher	50
The toBeFocused jQuery matcher	51
The toBeDisabled jQuery matcher	51
More matchers	51
Summary	51
Chapter 4: Asynchronous Testing – AJAX	53
Acceptance criterion	53
Setting up the scenario	55
Installing Node.js	55
Coding the server	55
Running the server	56
Writing the spec	57
Asynchronous setups and teardowns	57
Asynchronous specs	58
Timeout	59
Summary	60
Chapter 5: Jasmine Spies	61
The "bare" spy	61
Spying on an object's functions	62
Testing DOM events	63
Summary	64
Chapter 6: Light Speed Unit Testing	65
Jasmine stubs	65
Jasmine Ajax	67
Installing the plugin	67
A fake XMLHttpRequest	67
Summary	69
Chapter 7: Testing React Applications	71
Project setup	72
Our first React component	72
The Virtual DOM	75
JSX	76
Using JSX with Jasmine	78
Component attributes (props)	80
Component events	82
Component state	85
Component life cycle	89
Composing components	91
Summary	92

Chapter 8: Build Automation	93
Module bundler – webpack	94
Module definition	94
Webpack project setup	95
Managing dependencies with NPM	96
Webpack configuration	97
The spec runner	99
Testing a module	100
Test runner: Karma	100
Quick feedback loop	102
Watch and run the tests	102
Watch and update the browser	103
Optimizing for production	103
Static code analysis: JSHint	105
Continuous integration – Travis-CI	107
Adding a project to Travis-CI	107
Project setup	108
Summary	108
Index	109

Preface

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

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

Jasmine JavaScript Testing, Second Edition is a practical guide to writing and automating JavaScript testing for web applications. It uses technologies such as Jasmine, Node.js, and webpack.

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), passes through the concepts of maintainable browser code, and concludes with a full refactoring to a React.js application build on ECMA Script 6 modules 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 tests.

Chapter 2, Your First Spec, helps you learn the thought process behind thinking in terms of 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. Also demonstrated, is 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 terms of 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 an 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 Node.js 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 do behavior checking.

Chapter 6, Light Speed Unit Testing, helps you to learn about the issues with AJAX testing and how you can make your tests run faster using stubs or fakes.

Chapter 7, Testing React Applications, introduces you to React, a library to build user interfaces, and covers how you can use it to improve the concepts presented in *Chapter 3, Testing Frontend Code*, to create richer and more maintainable applications, of course, driven by tests.

Chapter 8, Build Automation, presents you with the power of automation. It introduces you to webpack, a bundling tool for frontend assets. You will start to think in terms of 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. Finally, you are going to learn about running your tests from a command line and how this can be used in a continuous integration environment with *Travis.ci*.

What you need for this book

Besides a browser and a text editor, the only requirement to run some of the examples, is Node.js 0.10.x.

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 that you have a basic knowledge of JavaScript and HTML.

Conventions

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

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "We can include other contexts through the use of the `include` directive."

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() {  
    expect(investment.stock).toBe(stock);  
  });  
});
```

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

```
# npm install --save-dev webpack
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Clicking the **Next** button moves you to the next screen."



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 disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail feedback@packtpub.com, and mention the book's title in 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 at 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 from your account at <http://www.packtpub.com> for all the Packt Publishing books you have purchased. 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 could 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/submit-errata>, 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.

To view the previously submitted errata, go to <https://www.packtpub.com/books/content/support> and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted 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

If you have a problem with any aspect of this book, you can contact us at questions@packtpub.com, and we will do our best to address the problem.

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 **Model View Controller (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 beginnings 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, which means there could be small differences or bugs between them.

Besides that, you have issues with the language itself. Brendan Eich developed JavaScript in just 10 days, under a lot of management pressure at 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.

Version 5 of the ECMA specification introduced the `Object.seal` function, which prevents further changes on any object once called. But its current support is not widespread; Internet Explorer, for example, only implemented it on its version 9.

Another problem, is with 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 aforementioned code results in `'11'`. But the main problem is in its inconsistency; on multiplication, a string is converted into 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 resulting value of the preceding example is `'1212'`, a string.

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

Jasmine and behavior-driven development

Jasmine is a little **behavior-driven development** (BDD) test framework created by the developers 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 right, starting with what a test unit is.

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 BDD, which is a rethink of **test-driven development** (TDD).

In traditional unit testing practice, the developer is left with loose guidelines on how to start 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 such as:

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

As shown in the following list, this acceptance criterion is written following an underlying pattern:

- **Given:** This provides an initial context
- **When:** This defines the event that occurs
- **Then:** This ensures an 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 preceding acceptance criterion written as a Jasmine test unit would be as follows:

```
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.

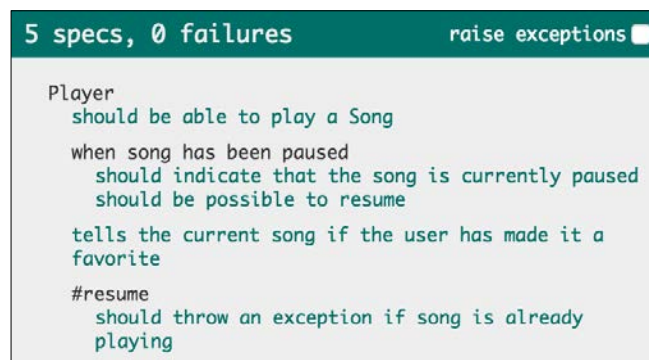
Downloading Jasmine

Getting started with Jasmine is actually pretty simple.

Open the Jasmine website at <http://jasmine.github.io/2.1/introduction.html#section-Downloads> and download the **Standalone Release** (version 2.1.3 is going to be used in the book).

While at the Jasmine website, you might notice that it is actually a live page executing the specs contained in 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):



This shows the `SpecRunner.html` file 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 src="lib/jasmine-2.1.3/jasmine.js"></script>
<script src="lib/jasmine-2.1.3/jasmine-html.js"></script>
<script src="lib/jasmine-2.1.3/boot.js"></script>
```

The runner references the source files:

```
<script type="text/javascript" src="src/Player.js"></script>
<script type="text/javascript" src="src/Song.js"></script>
```

The runner references a special `SpecHelper.js` file that contains code shared between specs:

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

The runner also references the spec files:

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



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.

The Jasmine framework is set up inside the `lib/jasmine-2.1.3/boot.js` file, and although it's an extensive file, most of its content is in documentation on how the setup actually happens. It is recommended that you open it in a text editor and study its content.

Although, for now, we are running the specs in the browser, in *Chapter 8, Build Automation*, we are going to make the same specs and code run on a **headless browser**, such as PhantomJS, and have 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.

And although not covered in this book, Jasmine can also be used to test server-side JavaScript code written for environments such as Node.js.

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

Summary

In this chapter, you saw 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 to 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, and we are going to guide you through how to write your first spec, think in test-first terms for development, and also show you 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 that you are developing an application to track investments in the stock market.

The following screenshot of the form illustrates how a user might create a new investment on this application:

Symbol:	Shares:	Share price:	
PETO	100	35	Add

This is a form to add investments

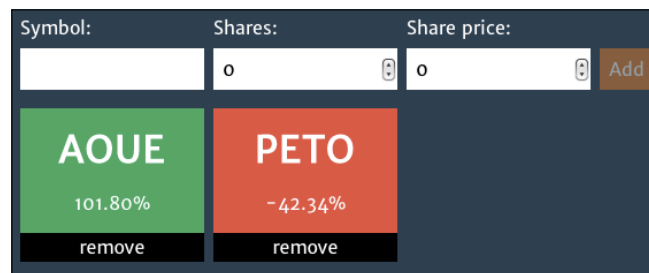
This form will allow the input of three values that define an investment:

- First, we will input **Symbol**, which represents which company (stock) the user is investing in
- Then, we will input how many **Shares** the user has bought (or invested in)
- Finally, we will input how much the user has paid for each share (**Share price**)

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 their other investments, as shown in the following screenshot:



This is a form and list of investments

The idea is to display how well their 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 indicates whether it is a good (profit) or a bad (loss) investment.

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

- One is in the `AOUE` stock, which is scoring a profit of `101.80%`
- Another is 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 price paid
- Given an investment, it should have a cost

Using the standalone distribution downloaded in the previous chapter, the first thing we need to do is 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 an `InvestmentSpec.js` file and add the following lines:

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

The `describe` function 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 (a collection of test cases). It accepts two parameters, which are as follows:

- The name of the test suite—in this case, `Investment`
- A function that 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 (or test case), 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, which are as follows:

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

To run this spec, add it to the runner, as follows:

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

Execute the spec by opening the runner on the browser. The following output can be seen:



This is the first spec's passing result on the browser

It 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** (or expectation) 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 using the global Jasmine function `expect`, along with a **matcher** that indicates what comparison must be made with the values.

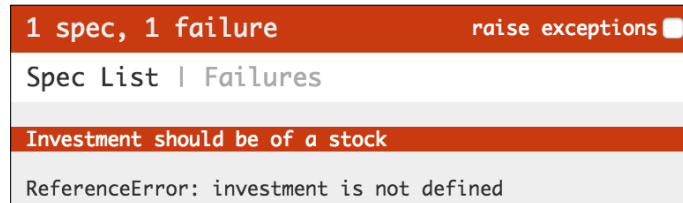
Regarding the current spec (it is expected that the investment is of a stock), in Jasmine this translates to the following code:

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

Add the preceding highlighted code to the `InvestmentSpec.js` file. 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, which in this case is `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 whether 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 spec that previously passed has now failed, as shown in the following screenshot:



This shows the first spec's failure results

This spec failed 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 in the `InvestmentSpec.js` file, as follows:

```
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, as follows:



Here the spec asks for an `Investment` class

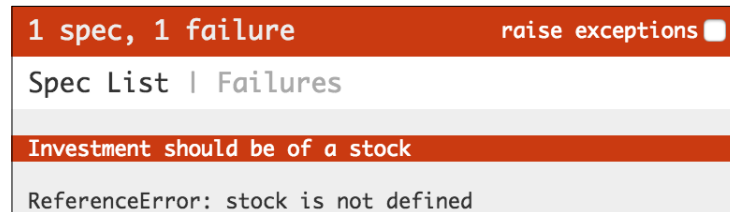
You can see that the error has changed to `Investment is not defined`. It now asks for the `Investment` function. So, create a new `Investment.js` file in the `src` folder and add it to the runner, as shown in the following code:

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

To define `Investment`, write the following constructor function in the `Investment.js` file inside the `src` folder:

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

This makes the error change. It now complains about the missing `stock` variable, as shown in the following screenshot:

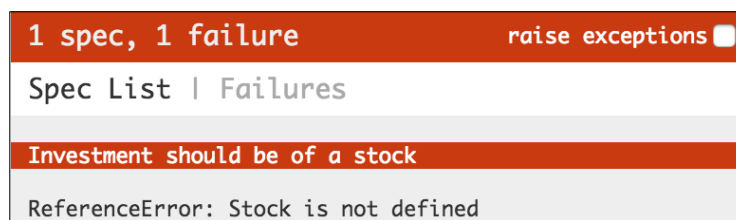


This shows a missing `stock` error

One more time, we feed the code it is asking for into the `InvestmentSpec.js` file, as shown in the following code:

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

The error changes again; this time it is about the missing `Stock` function:



Here the spec asks for a `Stock` class

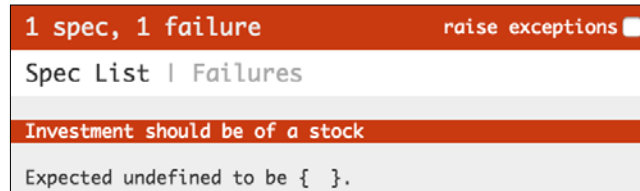
Create a new file in the `src` folder, name it `Stock.js`, and add it to the runner. Since the `Stock` function is going to be a dependency of `Investment`, we should add it just before `Investment.js`:

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

Write the `Stock` constructor function to the `Stock.js` file:

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

Finally, the error is about the expectation, as shown in the following screenshot:



The expectation is undefined to be `Stock`

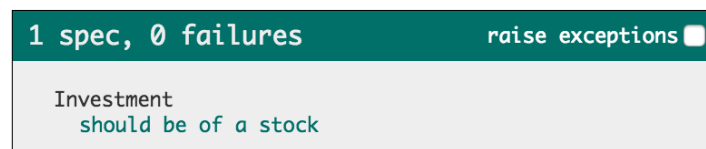
To fix this and complete this exercise, open the `Investment.js` file inside the `src` folder, and add the reference to the `stock` parameter:

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

In the spec file, pass `stock` as a parameter to the `Investment` function:

```
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 will have a passing spec:



This shows an `Investment` spec that passes

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 spec that has failed. You must not write code unless its purpose is to fix a failed spec.

Setup and teardown

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

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

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

```
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 changed 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.

Implementing this in the `Investment` function is pretty simple—instead of having multiple parameters on the function declaration, it 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, as shown here:

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

The code is now refactored. We can run the tests to see that only the new spec fails, as shown here:

2 specs, 1 failure	raise exceptions <input type="checkbox"/>
Spec List Failures	
Investment should have the invested shares quantity	
Expected undefined to equal 100.	

This shows the failing shares spec

To fix this, change the `Investment` constructor to make the assignment to the `shares` property, as follows:

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

Finally, everything on your screen is green:

2 specs, 0 failures	raise exceptions <input type="checkbox"/>
Investment should be of a stock should have the invested shares quantity	

This shows the passing shares spec

But as you can see, the following code, which instantiates `Stock` and `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 using the `beforeEach` function:

```
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);
  });
});
```

This looks 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 fulfill the expectation.

There is also a **teardown** function (`afterEach`) that sets the 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*.

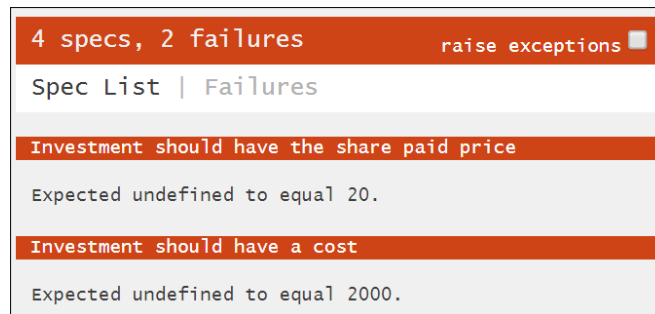
To finish the specification of `Investment`, add the remaining two specs to the `InvestmentSpec.js` file, inside the spec folder:

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

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

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

Run the specs to see them fail, as shown in the following screenshot:



This shows the failing cost and price specs

Add the following code to fix them in the `Investment.js` file inside the `src` folder:

```
function Investment (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 pass:

```
4 specs, 0 failures      raise exceptions ☐

Investment
  should be of a stock
  should have the invested shares quantity
  should have the share payed price
  should have a cost
```

This shows all four Investment specs passing



It is important to always see a spec fail before writing the code to fix it; otherwise, how would you know that you really need to 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 on investment (ROI)**
- Given an investment, when its stock share price valorizes, it should be a good investment

Both these criteria share the same behavior when the investment's stock share price valorizes.

To translate this into Jasmine, you can nest a call to the `describe` function inside the existing one in the `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 the setup and teardown functions (`beforeEach`, `afterEach`).

Setup and teardown

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

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

So, we can add a setup function to this new `describe` function that changes the share price of the stock, so that 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`:

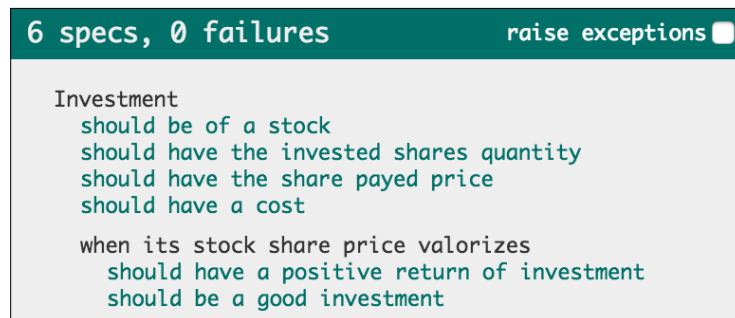
```
describe("Investment", function() {
  describe("when its stock share price valorizes", function() {
    beforeEach(function() {
      stock.sharePrice = 40;
    });
  });
});
```

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

After adding the missing functions to `Investment` in the `Investment.js` file:

```
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 pass:



This shows the nested describe specs pass

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` and `toEqual` matchers. These are the two base 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()).toEqual(true);
```

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

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

This creates a much better relation with the acceptance criterion:

So, here "should be a good investment" becomes "expect investment to be a good investment".

Implementing it is quite simple. You do so by calling the `jasmine.addMatchers` function—ideally inside a setup step (`beforeEach`).

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

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

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

The function being defined here is not the matcher itself but a factory function to build the matcher. Its purpose, once called is to return an object containing a compare function, as follows:

```
jasmine.addMatchers({  
  toBeAGoodInvestment: function () {  
    return {  
      compare: function (actual, expected) {  
        // matcher definition  
      }  
    };  
  }  
});
```


The `compare` function will contain the actual matcher implementation, and as can be observed by its signature, it receives both values being compared (the `actual` and `expected` values).

For the given example, the `investment` object will be available in the `actual` argument.

Then, Jasmine expects, as the result of this `compare` function, an object with a `pass` attribute with a boolean value `true` to indicate that the expectation passes and `false` if the expectation fails.

Let's have a look at the following valid implementation of the `toBeAGoodInvestment` matcher:

```
toBeAGoodInvestment: function () {
  return {
    compare: function (actual, expected) {
      var result = {};
      result.pass = actual.isGood();
      return result;
    }
  };
}
```

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

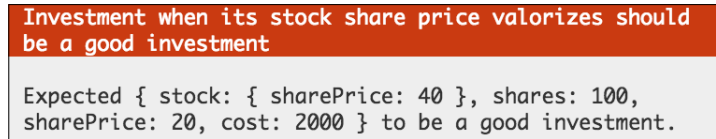
```
it("should be a good investment", function() {
  expect(investment).toBeAGoodInvestment();
});
```

After the change, the specs should still pass. 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 the `Investment.js` file, in the `src` folder to always return `false`:

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

When running the specs again, Jasmine generates an error message stating Expected { stock: { sharePrice: 40 }, shares: 100, sharePrice: 20, cost: 2000 } to be a good investment, as shown in the following screenshot:



Investment when its stock share price valorizes should be a good investment

Expected { stock: { sharePrice: 40 }, shares: 100, sharePrice: 20, cost: 2000 } to be a good investment.

This is the custom matcher's message

Jasmine does a great job generating this error message, but it also allows its customization via the `result.message` property of the object returned as the result of the matcher. Jasmine expects this property to be a string with the following error message:

```
toBeAGoodInvestment: function () {
  return {
    compare: function (actual, expected) {
      var result = {};
      result.pass = actual.isGood();
      result.message = 'Expected investment to be a good investment';
      return result;
    }
  };
}
```

Run the specs again and the error message should change:



Investment when its stock share price valorizes should be a good investment

Expected investment to be a good investment

This is the 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 the `InvestmentSpec.js` file inside the `spec` folder by adding new and nested `describe` and `spec`, as follows:

```
describe("when its stock share price devalorizes", function() {
  beforeEach(function() {
    stock.sharePrice = 0;
  });

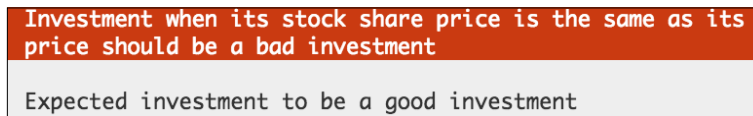
  it("should have a negative return of investment", function() {
    expect(investment.roi()).toEqual(-1);
  });

  it("should be a bad investment", function() {
    expect(investment).not.toBeAGoodInvestment();
  });
});
```

But there is a catch! Let's break the investment implementation in the `Investment.js` file code so that it is always a good investment, as follows:

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

After running the specs again, you can see that this new spec fails, but the error message, `Expected investment to be a good investment, is wrong`, as shown in the following screenshot:



This is the custom matcher's wrong custom negated message

That is the message that was hardcoded inside the matcher. To fix this, you need to make the message dynamic.

Jasmine only shows the message if the matcher fails, so the proper way of making this message dynamic is to consider what message is supposed to be shown when the given comparison is invalid:

```
compare: function (actual, expected) {
  var result = {};
  result.pass = actual.isGood();
```

```

    if (actual.isGood()) {
      result.message = 'Expected investment to be a bad investment';
    } else {
      result.message = 'Expected investment to be a good investment';
    }

    return result;
  }
}

```

This fixes the message, as shown in the following screenshot:



This shows the custom matcher's custom dynamic message

Now this matcher can be used anywhere.

Before continuing in the chapter, change the `isGood` method back again to its correct implementation:

```

Investment.prototype.isGood = function() {
  return this.roi() > 0;
};

```

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

```
expect(investment.cost).toBe(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 in the `SpecHelper.js` file, inside the `spec` folder, as follows:

```

beforeEach(function() {
  jasmine.addMatchers({
    toBe: function () {
      return {
        compare: function (actual, expected) {
          return actual === expected;
        }
      };
    }
  });
});

```

By implementing any matcher, check first whether there is one available that already does what you want.

For more information, check the official documentation at the Jasmine website http://jasmine.github.io/2.1/custom_matcher.html.

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 of how JavaScript handles type.

The `toEqual` built-in matcher

The `toEqual` matcher 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), as shown in the following code:

```
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 whether 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 that you use the `toEqual` operator in most cases and resort to the `toBe` matcher only when you want to check whether 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 to be 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 the following 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 whether 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();
  });
});
```



```
        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`, as shown in the following code:

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

As good practice, try to use these matchers instead of their `toBe` counterparts whenever possible.

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, as follows:

```
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 whether an array contains an element, or whether a string can be found inside another string. For these use cases, you can use the `toContain` matcher, as follows:

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

```
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` matchers are simple and used to perform numeric comparisons – something that is 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 toBeCloseTo built-in matcher

This is a special matcher used to compare floating-point numbers with a defined set of precision – something that is best explained by this example:

```
describe("toBeCloseTo", function() {
  it("should pass when the actual is closer with a given precision",
function() {
    expect(3.1415).toBeCloseTo(2.8, 0);
    expect(3.1415).not.toBeCloseTo(2.8, 1);
  });
});
```

The first parameter is the number being compared, and the second is the precision in the number of decimal cases.

The `toThrow` built-in matcher

Exceptions are a language's way of demonstrating 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 whether 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 is executed, and if it throws the `Some exception` exception, the test passes.

Summary

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

You got familiar with Jasmine matchers and know how powerful they are in describing a spec intent. You 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 at how we can use the concepts learned in this chapter to start testing web applications, which are 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, which 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, we are going to get the best practices on how to write maintainable and testable browser code.

To implement the user interface, we 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/2.1.0/lib/jasmine-jquery.js>
- <https://raw.githubusercontent.com/velesin/jasmine-jquery/2.1.0/vendor/jquery/jquery.js>

Save these files as `jasmine-jquery.js` and `jquery.js` respectively inside the `lib` folder, and add them to `SpecRunner.html`, as follows:

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

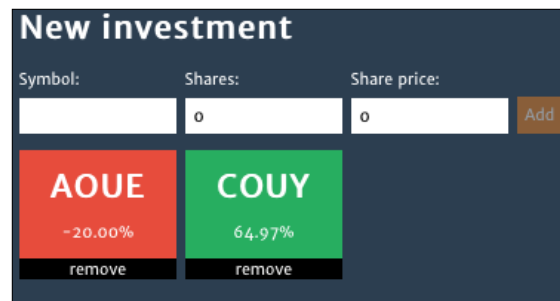
As seen until now, we have already created separate abstractions to handle both an investment and its associated stock. Now, it is time to develop this application's user interface and achieve a good result, which is all a matter of organization and good practices.

The same principles of software engineering that we apply on the server-side code must not be left behind when writing 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 code bases that plague most of the Web, which are code bases that are impossible to test. And the best way not to fall into this trap is by coding the application driven by tests.

Consider the mockup interface of our Investment Tracker application:



This shows the Investment Tracker application's mockup interface

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

- One responsibility is to add an investment
- Another responsibility is to list the added investments

So, we could start by breaking this interface into two different components. 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 break up the code, but how do we organize it? Until now, 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 until now and create a new file, `NewInvestmentView.js`, and place it in the `src` folder, as follows:

```
(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 until now. 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 new scope to declare 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 as a parameter. In this example, it is already passing three dependencies to the `NewInvestmentView` code: `jQuery`, `Investment`, and `Stock`.

You can see this at 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.

If we need to make anything global, we do that explicitly by attaching it with the global object, as follows:

```
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.

Although this practice does not have a great 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 use it throughout the rest of the 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, such as the following:

- `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 this is a good start.

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

```
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() {  
  });  
});
```

However, 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, we 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 to manually append the required DOM element inside a setup function, as follows:

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

Then, remove it during teardown, as follows:

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

Otherwise, the spec would append 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 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, as follows:

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

Change the code to the following:

```
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, as follows:

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

By default, the extension looks for files inside the `spec/javascripts/fixtures` folder (for the previous example, it would be `spec/javascripts/fixtures/MyFixture.html`) and loads its content inside a container, as follows:

```
<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)` function accepts a parameter with the content to be placed in the container:

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

The other available functions are as follows:

- `appendLoadFixtures(fixtureUrl[, fixtureUrl, ...])`: Instead of overwriting the content of the fixture container, this appends it
- `readFixtures(fixtureUrl[, fixtureUrl, ...])`: This reads a fixture container's content, but instead of appending it to the document, it returns a string with its contents
- `appendSetFixtures(html)`: This is the same as `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 at the test suite's 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, ...])` function loads one or more files in the cache without appending them to the document.

There is an issue, though, while using HTML. Jasmine jQuery loads the HTML fixtures using AJAX, but because of the **same origin policy (SOP)**, modern browsers will block 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, there is a workaround available in Chrome through the **command-line interface (CLI)** argument `--allow-file-access-from-files`.

As an example, in Mac OS X, it would require the following command in bash to open Chrome with this flag:

```
$ open "Google Chrome.app" --args --allow-file-access-from-files
```

More details on this issue can be seen at the GitHub ticket <https://github.com/velesin/jasmine-jquery/issues/4>.

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

Create a folder named `fixtures` inside the `spec` folder. Based on the mockup interface, we can create a new HTML fixture called `NewInvestmentView.html` inside the `fixtures` folder, as follows:

```
<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>
```

This is an HTML fixture because it would otherwise be rendered by a server and the JavaScript code would simply attach to it and add behavior.

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

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

In the `NewInvestmentSpec.js` file, add a 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 after the `Stock.js` and `Investment.js` files are added, as follows:

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

Basic View coding rules

Now, it is time to start coding the first View component. To help us 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 at the `NewInvestmentView.js` file, it uses jQuery to get the element for this selector and to store it in an instance variable `$element` (source), as follows:

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

To make sure this code works, we should write the following test for it in the `NewInvestmentViewSpec.js` file:

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

The `toExist` matcher is a custom matcher provided by the Jasmine jQuery extension to check whether an element exists in the document. It validates the existence of the property on the JavaScript object and also the successful association with the DOM element.

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

Another advantage of having an 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, as follows:

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

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

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

However, 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, as follows:

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

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

Since we will use this pattern everywhere inside the View code, we can create a function and use it instead, as shown in the following code:

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

Now let's 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 this:

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

However, 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 `NewInvestmentView`.

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

To fix this, the developer would need to look at the entire code base for references to that class name.

A much safer approach is to respect the View and use its APIs, as shown in the following code:

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

When implemented, that would look like the following:

```
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.

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

Integrating 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 about integrating `NewInvestmentView` and `InvestmentListView`?

You could write an acceptance criterion for `NewInvestmentView`, as follows:

Given the new investment View, when its add button is clicked, then it should add an investment to the list of investments.

This is 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, as follows:

```
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);
    });
  });
});
```

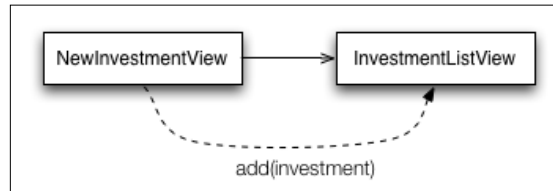
This solution creates a dependency between the two Views. The `NewInvestmentView` constructor now receives an instance of `InvestmentListView` as its `listView` parameter.

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

```
function NewInvestmentView (params) {
  this.listView = params.listView;

  this.$element.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:



This shows a 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.

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, we want to list investments in a table too. This would impose a change in `NewInvestmentView` to support both the list and table Views, as follows:

```
function NewInvestmentView (params) {
  this.listView = params.listView;
  this.tableView = params.tableView;

  this.$element.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, future-proof solution. Let's rewrite it as:

Given the Investment Tracker application, when a new investment is created, then it should add the investment to the list of investments.

We can see by the acceptance criterion that it has introduced a new subject to be tested: 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, as shown in the following code:

```
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);
  });
});
```

We can see the same setup code that once was inside the `NewInvestmentView` spec. It loads the fixtures required by both Views, instantiates both `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, we can see the function call to the `newView.create` function to create a new investment.

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

But how does the integration happen? We can see that 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 `newView`. This observer function will be invoked later when a new investment is created.

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

```
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.

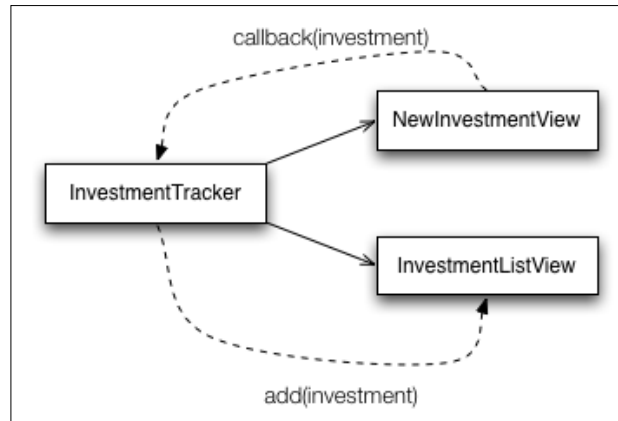
Although the prepended underline character won't actually change the visibility of the attribute, it at least informs a user of this object that the `_callback` attribute might change or even be removed in the future.

Later, when the `create` method is invoked, it invokes `_callback`, passing the new investment as a parameter, as follows:

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

A more complete implementation would need to allow multiple calls to `onCreate`, storing every passed callback.

Here is the solution illustrated for better understanding:



Using callbacks to integrate the two Views

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

Testing Views with jQuery matchers

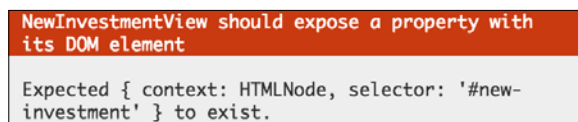
Besides its HTML fixture module, the Jasmine jQuery extension comes with a set of custom matchers, which help in writing expectations with the DOM elements.

The biggest advantage of using these custom matchers, as demonstrated, is that they generate better error messages. So, although we can write all specs without using any of these matchers, it would get us much more useful information when an error happens if we used the matchers.

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

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

If this spec fails, we get a nice error message, as shown in the following screenshot:

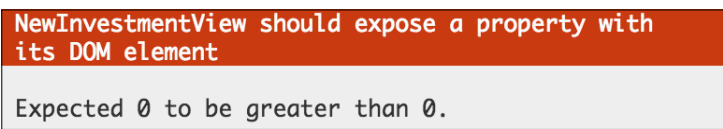


This shows a nice custom matcher error message

Now, we rewrite this spec without the custom matcher (still making the same validation):

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

This time, the error message gets less informative:



```
NewInvestmentView should expose a property with
its DOM element
Expected 0 to be greater than 0.
```

Upon reading the error, we 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 module.

The `toBeMatchedBy` jQuery matcher

This matcher checks whether the element matches the passed CSS selector, as follows:

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

The toContainHtml jQuery matcher

This matcher checks whether the content of the element matches the passed HTML, as follows:

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

The toContainElement jQuery matcher

This matcher checks whether the element contains any child element matching the passed CSS selector, as follows

```
it("should allow the input of the share price", function() {
  expect(view.$element).toContainElement('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 with the following code:

```
it("should have an empty stock symbol", function() {
  expect(view.$element.find('.new-investment-stock-symbol')).
  toHaveValue('');
});

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

The toHaveAttr jQuery matcher

This matcher tests whether 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.$element.find('.new-investment-share-price')).
  toHaveAttr('value', '0');
});
```

The toBeFocused jQuery matcher

The following code illustrates how the matcher checks whether the input element is focused:

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

The toBeDisabled jQuery matcher

This matcher checks whether the element is disabled with the following code:

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

More matchers

The extension has many more available matchers; make 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 saw 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 also showed you how to test code that interacts with the browser's DOM by the use of custom jQuery matchers.

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://localhost/data.json', {  
  success: function (data) {  
    // handle the result  
  }  
});
```

In this chapter, we are going to learn the different ways Jasmine allows us to write tests for asynchronous code.

Acceptance criterion

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

Stock when fetched, should update its share price

Using the techniques we have showed you until now, you could write this acceptance criterion in `StockSpec.js`, inside the `spec` folder file, as follows:

```
describe("Stock", function() {  
  var stock;  
  var originalSharePrice = 0;
```



```
beforeEach(function() {
  stock = new Stock({
    symbol: 'AOUE',
    sharePrice: originalSharePrice
  });
});

it("should have a share price", function() {
  expect(stock.sharePrice).toEqual(originalSharePrice);
});

describe("when fetched", function() {
  var fetched = false;
  beforeEach(function() {
    stock.fetch();
  });

  it("should update its share price", function() {
    expect(stock.sharePrice).toEqual(20.18);
  });
});
});
```

That would lead to the implementation of the `fetch` function from the `Stock.js` file inside the `src` folder, as follows:

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

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

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

```
{
  "sharePrice": 20.18
}
```

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 for a server. And in *Chapter 8, Build Automation*, we are going to see how to use Node.js as a foundation for an advanced build system.

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. Perform the following steps to install Node.js:

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

To check other installation methods and instructions on how to install Node.js 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 the `node` and `npm` commands 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 the following code to it:

```
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);
```

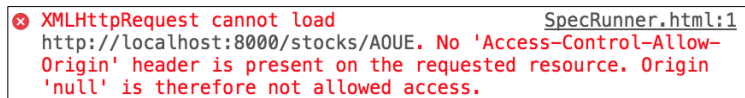
To handle the HTTP requests, we use **Express**, a Node.js web application framework. By reading the code, you can see that it defines a route to `/stocks/:symbol`, so it accepts requests such as `http://localhost:8000/stocks/AQOE` and responds with JSON data.

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

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

This issue was first demonstrated while using HTML fixtures in *Chapter 3, Testing Frontend Code*.

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):



This shows the 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 following `node` command:

```
$ node server.js
```

To check whether it is working, hit `http://localhost:8000/stocks/AOUE` on your browser, and you should receive the JSON response:

```
{"sharePrice": "20.18"}
```

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://localhost: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. This is 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(20.18);  
});
```

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.

Asynchronous setups and teardowns

In the example shown, we invoke the `fetch` function during the spec's setup (the `beforeEach` function).

The only thing we need to do to identify that this setup step is asynchronous is add a `done` argument to its function definition:

```
describe("when fetched", function() {  
  beforeEach(function(done) {  
  
  });  
  
  it("should update its share price", function() {  
    expect(stock.sharePrice).toEqual(20.18);  
  });  
});
```

Once Jasmine identifies this `done` argument, it passes as its value a function that must be called once the asynchronous operation is completed.

So we could then pass this `done` function as a success callback of the `fetch` function:

```
beforeEach(function(done) {  
  stock.fetch({  
    success: done  
  });  
});
```

At the implementation, invoke it once the AJAX operation is completed:

```
Stock.prototype.fetch = function(params) {  
  params = params || {};  
  var that = this;  
  var success = params.success || function () {};  
  var url = 'http://localhost:8000/stocks/'+that.symbol;  
  
  $.getJSON(url, function (data) {  
    that.sharePrice = data.sharePrice;  
    success(that);  
  });  
};
```

That is all there is to it; Jasmine will wait for the AJAX operation to be completed and the test will pass.

When required, it is also possible to have asynchronous `afterEach` definitions using the same `done` argument.

Asynchronous specs

Another approach would be to have an asynchronous spec instead of an asynchronous setup. To demonstrate how this would work, we are going to need to rewrite our previous acceptance criteria:

```
describe("Stock", function() {  
  var stock;  
  var originalSharePrice = 0;  
  
  beforeEach(function() {  
    stock = new Stock({  
      symbol: 'AOUE',  
      sharePrice: originalSharePrice
```

```
    });  
  });  
  
  it("should be able to update its share price", function(done) {  
    stock.fetch();  
    expect(stock.sharePrice).toEqual(20.18);  
  });  
});
```

Again, all we have to do is add a `done` argument to its function definition and invoke the `done` function once the test is done:

```
it("should be able to update its share price", function(done) {  
  stock.fetch({  
    success: function () {  
      expect(stock.sharePrice).toEqual(20.18);  
      done();  
    }  
  });  
});
```

The difference here is that we had to move the expectation for it to be inside the `success` callback right before invoking the `done` function.

Timeout

When writing asynchronous specs, Jasmine will wait for 5 seconds, by default, for the `done` callback to be called, failing the spec if it is not called before this timeout.

In this contrived example, where the server was a simple stub returning static data, that timeout was not a problem, but there are situations where that default time is not enough to complete an asynchronous task.

Although it is not recommended to have long-running specs, it is nice to know there is a way around this default behavior by changing a simple configuration variable in Jasmine called `jasmine.DEFAULT_TIMEOUT_INTERVAL`.

To make it take effect in the entire suite, one could set it at the `SpecHelper.js` file, as follows:

```
beforeEach(function() {  
  jasmine.DEFAULT_TIMEOUT_INTERVAL = 10000;  
  
  jasmine.addMatchers({  
    // matchers code
```

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

To make it take effect over a single spec, change its value in `beforeEach` and restore it during `afterEach`:

```
describe("Stock", function() {  
  var defaultTimeout;  
  
  beforeEach(function() {  
    defaultTimeout = jasmine.DEFAULT_TIMEOUT_INTERVAL;  
    jasmine.DEFAULT_TIMEOUT_INTERVAL = 10000;  
  });  
  
  afterEach(function() {  
    jasmine.DEFAULT_TIMEOUT_INTERVAL = defaultTimeout;  
  });  
  
  it("should be able to update its share price", function(done) {  
  
    });  
  });  
});
```

Summary

In this chapter, you have seen how to test asynchronous code, a scenario common when testing server interactions (AJAX).

I have also presented you with 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.

In the next chapter, we are going to learn about spies and how we can use them to perform behavior checking.

5

Jasmine Spies

A 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 whether a test expectation was a success. 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 the `NewInvestmentView` test suite:

```
describe("NewInvestmentView", function() {
  var view;

  // setup and other specs ...

  describe("with its inputs correctly filled", function() {

    // setup and other specs ...

    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 created
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`). A Jasmine spy is a special kind of function that tracks calls and arguments made to it.

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 `toHaveBeenCalled` and `toHaveBeenCalledWith` matchers to check whether the `callbackSpy` was called and with the right parameters (`investment`), thereby making a behavior check.

Spying on an object's functions

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

Consider the following example, which is aimed at validating that when the form is submitted, the `create` function of `view` has to be called:

```
describe("NewInvestmentView", function() {
    var view;

    // setup and other specs ...

    describe("with its inputs correctly filled", function() {

        // setup and other specs ...
```

```
describe("and when the form is submitted", function() {
  beforeEach(function() {
    spyOn(view, 'create');
    view.$element.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` with a spy.

Then, later in 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 intend to write a spec that checks whether an event is being triggered.

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

We can write a new acceptance criterion to the `NewInvestmentView` test suite to check that its form is being submitted when we click on the add button:

```
describe("and when its add button is clicked", function() {
  beforeEach(function() {
    spyOnEvent(view.$element, 'submit');
    view.20.18.find('input[type=submit]').click();
  });

  it("should submit the form", function() {
    expect('submit').toHaveBeenTriggeredOn(view.20.18);
  });
});
```

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

It works by accepting `view.20.18`, which is a DOM element, and the `submit` event we want to spy on. Then, later on, we use the jasmine jQuery matcher `toHaveBeenTriggeredOn` to check 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 at how we can use fakes and stubs to replace real dependencies of our specs and speed up their execution.

6

Light Speed Unit Testing

In *Chapter 4, Asynchronous Testing – AJAX*, we saw how including AJAX testing in the application can increase the complexity of the tests. In the example in that chapter, we created a server where the results were predictable. It was basically a complex test fixture. 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 transmit, which hurts the quick feedback loop that unit testing usually provides.

You can also say that these specs test both the client and the server code and, therefore, could not be considered unit tests; rather, they could be considered 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.

We know that the stock's `fetch` function is implemented using the `$.getJSON` function, as follows:

```
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 with the following code:

```
describe("when fetched", function() {
  beforeEach(function() {
    spyOn($, 'getJSON').and.callFake(function(url, callback) {
      callback({ sharePrice: 20.18 });
    });
    stock.fetch();
  });

  it("should update its share price", function() {
    expect(stock.sharePrice).toEqual(20.18);
  });
});
```

But this time, we will use the `and.callFake` 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: 20.18 }`).

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 a Jasmine plugin called **jasmine-ajax**, is to stub the browser's AJAX infrastructure instead, so the implementation of the AJAX request is free to be done in different manners.

Jasmine Ajax

Jasmine Ajax is an official plugin developed to help out the testing of AJAX requests. It changes the browser's AJAX request infrastructure to a fake implementation.

This fake (or mocked) implementation, although simpler, still behaves like the real implementation to any code using its API.

Installing the plugin

Before we dig into the spec implementation, first we need to add the plugin to the project. Go to <https://github.com/jasmine/jasmine-ajax/> and download the current release (which should be compatible with the Jasmine 2.x release). Place it inside the `lib` folder.

It is also needed to be added to the `SpecRunner.html` file, so go ahead and add another script:

```
<script type="text/javascript" src="lib/mock-ajax.js"></script>
```

A fake XMLHttpRequest

Whenever you are using jQuery to make AJAX requests, under the hood it is actually using the `XMLHttpRequest` object to 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 change the `XMLHttpRequest` object with a fake implementation. That is exactly what this plugin does.

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

```
describe("when fetched", function() {
  beforeEach(function() {
    jasmine.Ajax.install();
  });

  beforeEach(function() {
    stock.fetch();

    jasmine.Ajax.requests.mostRecent().respondWith({
      'status': 200,
      'contentType': 'application/json',
```

```
        'responseText': '{ "sharePrice": 20.18 }'
    });

    afterEach(function() {
        jasmine.Ajax.uninstall();
    });

    it("should update its share price", function() {
        expect(stock.sharePrice).toEqual(20.18);
    });
});
```

Drilling the implementation down:

1. First, we tell the plugin to replace the original implementation of the `XMLHttpRequest` object by a fake implementation using the `jasmine.Ajax.install` function.
2. We then invoke the `stock.fetch` function, which will invoke `$.getJSON`, creating `XMLHttpRequest` anew under the hood.
3. And finally, we use the `jasmine.Ajax.requests.mostRecent().respondWith` function to get the most recently made request and respond to it with a fake response.

We use the `respondWith` function, which accepts an object with three properties:

1. The `status` property to define the HTTP status code.
2. The `contentType` (JSON in the example) property.
3. The `responseText` property, which is a text string containing the response body for the request.

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

```
it("should update its share price", function() {
    expect(stock.sharePrice).toEqual(20.18);
});
```

Since the plugin changes the global `XMLHttpRequest` object, you must remember to tell Jasmine to restore it to its original implementation after the test runs; otherwise, you could interfere with the code from other specs (such as the Jasmine jQuery fixtures module). Here's how you can accomplish this:

```
afterEach(function() {
    jasmine.Ajax.uninstall();
});
```

There is also a slightly different approach to write this spec; here, the request is first stubbed (with the response details) and the code to be exercised is executed later.

The previous example is changed to the following:

```
beforeEach(function() {  
  jasmine.Ajax.stubRequest('http://localhost:8000/stocks/AOUE').  
  andReturn({  
    'status': 200,  
    'contentType': 'application/json',  
    'responseText': '{ "sharePrice": 20.18 }'  
  });  
  
  stock.fetch();  
});
```

It is possible to use the `jasmine.Ajax.stubRequest` function to stub any request to a specific request. In the example, it is defined by the URL `http://localhost:8000/stocks/AOUE`, and the response definition is as follows:

```
{  
  'status': 200,  
  'contentType': 'application/json',  
  'responseText': '{ "sharePrice": 20.18 }'  
}
```

The response definition follows the same properties as the previously used `respondWith` function.

Summary

In this chapter, you 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 two different ways in which you could test AJAX requests with a simple Jasmine stub and with the more advanced, fake implementation of the `XMLHttpRequest`.

You also got more familiar with spies and stubs and should be more comfortable using them in different scenarios.

In the next chapter, we are going to go further into the complexity of our application, and we will do an overall refactoring to transform it into a fully featured single-page application using the `React.js` library.

7

Testing React Applications

As a web developer, you are familiar with the way most websites are built today. There is usually a web server (in languages such as Java, Ruby, or PHP) that processes user requests and responds with markup (HTML).

This means that on every request, the web server interprets the user action through the URL and renders the entire page.

In an attempt to improve the user experience, more and more functionality started to get pushed from the server side to the client side, and JavaScript was no longer simply adding behavior to the page but was rendering it entirely. The biggest advantage was that a user action was no longer triggering a whole page refresh; the JavaScript code could deal with the entire browser document and mutate it accordingly.

Although this did improve the user experience, it started to add a lot of complexity to the application code, which led to increased maintenance costs and the worst – bugs in the form of inconsistencies between different parts of the screen.

In an attempt to bring sanity to this scenario, a number of libraries and frameworks were built, but they all failed in the sense that they didn't tackle the root cause of the entire problem – mutability.

Server-side rendering was easy because there was no mutation to deal with. Given a new application state, the server would simply render everything again. What if we could get benefits from this approach in our client-side JavaScript code?

That is exactly what **React** proposes to do. You declaratively write the interface code in the form of components and tell React to render. On any change of the application state, you can simply tell React to re-render again; it will then calculate the mutations required to move the DOM to the required state and apply them for you.

During this chapter, we are going to understand how React works by refactoring the code we've built so far into an SPA.

Project setup

However, before we can dive into React, first we need a small setup in our project to allow us to create React components.

Go to <http://facebook.github.io/react/downloads.html> and download the React Starter Kit Version 0.12.2 or higher.

After the download, you can unpack its contents and move all the files from within the build folder to our application's lib folder. Then, just load the React library onto the `SpecRunner.html` file.

```
<script src="lib/react-with-addons.js"></script>
<script src="lib/jquery.js"></script>
```

With the setup complete, we can move on to writing our very first component.

Our first React component

As stated in the introduction of this chapter, with React, you declaratively write the interface code through components.

The concept of a React component is analogous to the component concept presented in *Chapter 3, Testing Frontend Code*, so expect to see some similarities next.

With that in mind, let's create our very first component. To better understand what a React component is, we are going to use a very simple acceptance criterion and as usual start from the spec.

Let's implement "InvestmentListItem should render". It's very simple and not really *feature oriented* but is a good example to get us started.

With what we learned in *Chapter 3, Testing Frontend Code*, we could start coding this spec by creating a new file called `InvestmentListItemSpec.js` and save it in the `components` folder inside the `spec` folder:

```
describe("InvestmentListItem", function() {

  beforeEach(function() {
    // render the React component
  });

});
```

```
it("should render", function() {  
  expect(component.$el).toEqual('li.investment-list-item');  
});  
});
```

Add the new file to the `SpecRunner.html` file, as already demonstrated in previous chapters.

At the spec, we are basically using the `jasmine-jquery` plugin to expect that the encapsulated DOM element of our component is equal to a specific CSS selector.

How would we change this example to be a test of a React component? The only difference is the API to get the DOM node. Instead of `$element` with a jQuery object, React exposes a function called `ReactDOMNode()` that returns what it states—a DOM node.

With that, we can use the same assertion as before and have our test ready, as follows:

```
it("should render", function() {  
  expect(component.getDOMNode()).toEqual('li.investment-list-item');  
});
```

That was easy! So, the next step is to create the component, render it, and attach it to the document. That is simple as well; take a look at the following gist:

```
describe("InvestmentListItem", function() {  
  var component;  
  
  beforeEach(function() {  
    setFixtures('<div id="application-container"></div>');  
    var container = document.getElementById('application-container');  
  
    var element = React.createElement(InvestmentListItem);  
    component = React.render(element, container);  
  });  
  
  it("should render", function() {  
    expect(component.getDOMNode()).toEqual('li.investment-list-item');  
  });  
});
```

It might seem like a lot of code, but half of it is just boilerplate to set up a document element fixture that we can render the React component in:

1. First, we use the `setFixtures` function from `jasmine-jquery` to create an element in the document with the `application-container` ID. Then, using the `getElementById` API, we query for this element and save it in the `container` variable. The next two steps are the ones specific to React:

1. First, in order to use a component, we must first create an element from its class; this is done through the `React.createElement` function, as follows:

```
var element = React.createElement(InvestmentListItem);
```

2. Next, with the element instance, we can finally tell React to render it through the `React.render` function, as follows:

```
component = React.render(element, container);
```

3. The render function accepts the following two parameters:

- The React element
- A DOM node to render the element in

2. As of now, the spec is complete. You can run it and see it fail, showing the following error:

```
ReferenceError: InvestmentListItem is not defined.
```

3. The next step is to code the component. So, let's feed the spec, create a new file in the `src` folder, name it `InvestmentListItem.js`, and add it to the spec runner. This file should follow the module pattern we've been using until now.

4. Then, create a new class of component using the `React.createClass` method:

```
(function (React) {  
  var InvestmentListItem = React.createClass({  
    render: function () {  
      return React.createElement('li', { className: 'investment-  
list-item' }, 'Investment');  
    }  
  });  
  
  this.InvestmentListItem = InvestmentListItem;  
})(React);
```

5. At the least, the `React.createClass` method expects a single render function that should return a tree of React elements.
6. We use again the `React.createElement` method to create the element that is going to be the root of the rendering tree, as follows:

```
React.createElement('li', { className: 'investment-list-item' },  
  'Investment')
```

The difference from its previous usage in the `beforeEach` block is that here, it is also passing a list of **props** (with `className`) and a single child containing the text `Investment`.

We will get deeper into the meaning of the props parameter, but you can think of it as analogous to the attributes of an HTML DOM element. The `className` prop will turn into the class HTML attribute of the `li` element.

The `React.createElement` method signature accepts three arguments:

- The type of the component that can be either a string representing a real DOM element (such as `div`, `h1`, `p`) or a React component class
- An object containing the props values
- And a variable number of children components, which in this case, is just the `Investment` string

On rendering this component (by invoking the `React.render()` method), the result will be:

```
<li class="investment-list-item">Investment</li>
```

This is a direct representation of the JavaScript code that generated it:

```
React.createElement('li', { className: 'investment-list-item' },  
  'Investment');
```

Congratulations! You've built your first fully tested React component.

The Virtual DOM

When you define a component's render method and invoke the `React.createElement` method, you are not actually rendering anything in the document (you are not even creating DOM elements).

It is only through the `React.render` function that the representation created by invoking these `React.createElement` calls are effectively converted into real DOM elements and attached to the document.

This representation, defined by `ReactElements`, is what React calls the Virtual DOM. And `ReactElement` must not be confused with DOM elements; it is instead a light, stateless, immutable, virtual representation of a DOM element.

So why did React get into the trouble of creating a new way of representing the DOM? The answer here is *performance*.

As browsers evolved, JavaScript performance kept getting better and better, and today's application bottlenecks aren't actually JavaScript. You've probably heard that you should try touching the DOM as little as possible, and React allows you to do that by letting you interact with its own version of the DOM.

However, that is not the only reason. React has built a very powerful diffing algorithm that can compare two distinct representations of the Virtual DOM, compute their differences, and with that information, create mutations that then get applied to the real DOM.

It allows us to get back to the flow we used to have with server-side rendering. We can basically, on any change of the application state, ask React to re-render everything, and it will then compute the minimal number of changes required and apply only that to the real DOM.

It frees us developers from worrying about mutating the DOM and empowers us to write our user interfaces in a declarative way, while reducing bugs and improving productivity.

JSX

If you have any experience writing frontend JavaScript applications, you might be familiar with a few template languages. At this moment, you might be wondering where can you use your favorite template language (such as Handlebars) with React. And the answer is that you can't.

React doesn't make any distinction between markup and logic; in a React component, they are effectively the same.

However, what happens when we start crafting more complicated components? How would the form we built in *Chapter 3, Testing Frontend Code*, translate into a React component?

To just render it without any other logic, it would take a bunch of `React.createElement` calls, as follows:

```
var NewInvestment = React.createClass({
  render: function () {
```

```

    return React.createElement("form", {className: "new-investment"},
      React.createElement("h1", null, "New investment"),
      React.createElement("label", null,
        "Symbol:",
        React.createElement("input", {type: "text", className: "new-
investment-stock-symbol", maxLength: "4"})
      ),
      React.createElement("label", null,
        "Shares:",
        React.createElement("input", {type: "number", className: "new-
investment-shares"})
      ),
      React.createElement("label", null,
        "Share price:",
        React.createElement("input", {type: "number", className: "new-
investment-share-price"})
      ),
      React.createElement("input", {type: "submit", className: "new-
investment-submit", value: "Add"})
    );
  }
});

```

This is very verbose and hard to read. So, given that a React component is both markup and logic, wouldn't it be better if we could write it as a mixture of HTML and JavaScript? Here's how:

```

var NewInvestment = React.createClass({
  render: function () {
    return <form className="new-investment">
      <h1>New investment</h1>
      <label>
        Symbol:
        <input type="text" className="new-investment-stock-symbol"
maxLength="4" />
      </label>
      <label>
        Shares:
        <input type="number" className="new-investment-shares" />
      </label>
      <label>
        Share price:
        <input type="number" className="new-investment-share-price" />
      </label>
    </form>
  }
});

```



```
        <input type="submit" className="new-investment-submit"
value="Add" />
      </form>;
    }
  });
```

That is **JSX**, a JavaScript syntax extension that looks like XML and was built to be used with React.

It transforms into JavaScript, so given the latter example, it would compile directly to the plain JavaScript code presented before.

An important feature of the transformation process is that it doesn't change the line numbers; so, *line 10* in the JSX will translate into *line 10* in the transformed JavaScript file. This helps while debugging the code and doing static code analysis.

For more information about the language, you can check the official specification at <http://facebook.github.io/jsx/>, but for now, you can just follow the next examples as we dive into the features of this language.

It is important to know that it is not a requirement to use JSX while implementing React components, but it makes the process a lot easier. With that in mind, we are going to keep using it for now.

Using JSX with Jasmine

In order for us to use JSX with our Jasmine runner, there are a few changes we need to make.

First, we need to rename the files with which we want to use the JSX syntax to `.jsx`. Although this is not a requirement, it allows us to easily identify when a file is using this special syntax.

Next, on the `SpecRunner.html` file, we need to change the script tags to indicate that these are not regular JavaScript files, as follows:

```
<script src="src/components/InvestmentListItem.jsx" type="text/jsx"></script>
<script src="spec/components/InvestmentListItemSpec.jsx" type="text/jsx"></script>
```

Unfortunately, these are not the only changes we need to make. The browser doesn't understand JSX syntax, so we need to load a special transformer that will first transform these files into regular JavaScript.

This transformer comes bundled in the React starter kit, so just load it right after loading React, as follows:

```
<script src="lib/react-with-addons.js"></script>
<script src="lib/JSXTransformer.js"></script>
```

With this setup done, we should be able to run the tests, shouldn't we? Unfortunately, there is one more step.

If you try to open the `SpecRunner.html` file in the browser, you will see that the tests of `InvestmentListItem` are not being executed. That is because the transformer works by loading the script files through AJAX, transforming them and finally attaching them to the DOM. By the time this process is complete, Jasmine has already run the tests.

We need a way to inform Jasmine to wait for these files to load and be transformed.

The simplest way to do that is to change Jasmine's `boot.js` file placed in the `jasmine-2.1.3` folder, inside the `lib` folder.

In the original file, you are going to need to find the line that contains the `env.execute()` method and comment it out. It should be something like the following code:

```
window.onload = function() {
  if (currentWindowOnload) {
    currentWindowOnload();
  }
  htmlReporter.initialize();

  // delays execution so that JSX files can be loaded
  // env.execute();
};
```

Everything else in the file should remain the same. After this change, you will see that the tests are no longer running—none of them.

The only missing piece is invoking this `execute` method once the JSX files are loaded. To do so, we are going to create a new file called `boot-exec.js` inside the `jasmine.2.1.3` folder with the following content:

```
/**
 * Custom boot file that actually runs the tests.
 * The code below was extracted and commented out from the original
 * boot.js file.
 */
(function() {
```

```
var env = jasmine.getEnv();
env.execute();

}());
```

As you see, it is only executing the previously commented code from the original boot file.

To run this custom boot is very simple. We add it to the last line of the `<head>` tag of `SpecRunner.html` as a JSX type:

```
<!-- After all JSX files were loaded and processed, the tests can
finally run -->
<script src="lib/jasmine-2.1.3/boot-exec.js" type="text/jsx"></
script>

</head>
```

The `JSXTransformer` library guarantees that the scripts are loaded in the order they are declared. So, by the time the `boot-exec.js` file is loaded, the source and test files are already loaded.

With that, our test runner now supports JSX.

Component attributes (props)

Props are the way to pass down data from a parent to a child component in React.

For this next example, we want to change the `InvestmentListItem` component to render the value of the `roi` variable formatted in percentage.

To implement the next specs, we are going to use a few helper methods that React offers through the `React.addons.TestUtils` object, as follows:

```
describe("InvestmentListItem", function() {
  var TestUtils = React.addons.TestUtils;

  describe("given an Investment", function() {
    var investment, component;

    beforeEach(function() {
      investment = new Investment({
        stock: new Stock({ symbol: 'peto', sharePrice: 0.25 }),
        shares: 100,
        sharePrice: 0.20
```

```
    });

    component = TestUtils.renderIntoDocument(
      <InvestmentListItem investment={investment}/>
    );
  });

  it("should render the return of investment as a percentage",
function() {
    var roi = TestUtils.findRenderedDOMComponentWithClass(component,
'roi');
    expect(roi.getDOMNode()).toHaveText('25%');
  });
});
});
```

As you can see, we are no longer using the `setFixture` method from the `jasmine-jquery` matcher. Instead, we are using the `TestUtils` module to render the component.

The biggest difference here is that `TestUtils.renderIntoDocument` doesn't actually render in the document, but it renders into a detached node.

The next thing you will notice is that the `InvestmentListItem` component has an attribute (actually called **prop**) where we pass down `investment`.

Then, at the spec, we are using another helper method called `findRenderedDOMComponentWithClass` to look for a DOM element in our component variable.

This method returns `ReactDOM`. And again, we will use the `getDOMNode` method to get the actual DOM element and then use the `jasmine-jquery` matcher to check for its text value, as follows:

```
var roi = TestUtils.findRenderedDOMComponentWithClass(component,
'roi');
expect(roi.getDOMNode()).toHaveText('25%');
```

Implementing this behavior in the component is actually very simple:

```
(function (React) {
  var InvestmentListItem = React.createClass({
    render: function () {
      var investment = this.props.investment;

      return <li className="investment-list-item">
        <article>
```

```
        <span className="roi">{formatPercentage(investment.roi())}</span>
      span>
        </article>
      </li>;
    }
  });

  function formatPercentage (number) {
    return (number * 100).toFixed(0) + '%';
  }

  this.InvestmentListItem = InvestmentListItem;
})(React);
```

We can access any props passed to a component through the `this.props` object.

Expanding the original implementation, we've added a `span` element with the expected class from the spec.

To allow the return on investment to be dynamic, JSX has a special syntax. Using `{}`, you can invoke any JavaScript code in the middle of the XML. We are invoking the `formatPercentage` function while passing the `investment.roi()` value, as follows:

```
<span className="roi">{formatPercentage(investment.roi())}</span>
```

Again, just to make this clear, this JSX transformed into JavaScript would be:

```
React.createElement("span", {className: "roi"},
  formatPercentage(investment.roi())
```

It is important to know that a prop should be immutable. It is not the responsibility of a component to change its own prop values. You can consider a React component that has only props as a pure function, in that it always returns the same result value given the same argument values.

This makes testing very simple as there are no mutations or changes in the state to test a component.

Component events

UI applications have user events; in the web, they came in the form of DOM events. As React wraps each DOM element into React elements, handling them will be a little different yet very familiar.

For this next example, let's suppose that our application will allow users to delete an investment. We could write this requirement through the following acceptance criterion:

Given an investment, `InvestmentListItem` should notify an observer `onClickDelete` when the delete button is clicked on.

The idea here is the same as presented in the *Integrate Views with observers* section of *Chapter 3, Testing Frontend Code*.

So, how should we set the observer in a component? As we've already seen previously, **props** are the way to pass attributes to our component, as follows:

```
describe("InvestmentListItem", function() {
  var TestUtils = React.addons.TestUtils;

  describe("given an Investment", function() {
    var investment, component, onClickDelete;

    beforeEach(function() {
      investment = new Investment({
        stock: new Stock({ symbol: 'peto', sharePrice: 0.25 }),
        shares: 100,
        sharePrice: 0.20
      });

      onClickDelete = jasmine.createSpy('onClickDelete');

      component = TestUtils.renderIntoDocument(
        <InvestmentListItem investment={investment}
onClickDelete={onClickDelete}/>
      );
    });

    it("should notify an observer onClickDelete when the delete button
is clicked", function() {
      var deleteButton = TestUtils.findRenderedDOMComponentWithTag(component, 'button');
      TestUtils.Simulate.click(deleteButton);
      expect(onClickDelete).toHaveBeenCalled();
    });
  });
});
```

As you can see, we passed down another prop to the `onClickDelete` component, and as its value, we set a Jasmine spy, as follows:

```
onClickDelete = jasmine.createSpy('onClickDelete');

component = TestUtils.renderIntoDocument(
  <InvestmentListItem investment={investment}
  onClickDelete={onClickDelete}/>
);
```

Then, we found the delete button through its tag and used the `TestUtils` module to simulate a click, expecting the previously created spy to be called, as follows:

```
var deleteButton = TestUtils.findRenderedDOMComponentWithTag(component, 'button');
TestUtils.Simulate.click(deleteButton);
expect(onClickDelete).toHaveBeenCalled();
```

The `TestUtils.Simulate` module contains helper methods to simulate all types of DOM events, as follows:

```
TestUtils.Simulate.click(node);
TestUtils.Simulate.change(node, {target: {value: 'Hello, world'}});
TestUtils.Simulate.keyDown(node, {key: "Enter"});
```

Then, we got back to the implementation:

```
(function (React) {
  var InvestmentListItem = React.createClass({
    render: function () {
      var investment = this.props.investment;
      var onClickDelete = this.props.onClickDelete;

      return <li className="investment-list-item">
        <article>
          <span className="roi">{formatPercentage(investment.roi())}</span>
          <button className="delete-investment"
            onClick={onClickDelete}>Delete</button>
        </article>
      </li>;
    }
  });

  function formatPercentage (number) {
    return (number * 100).toFixed(0) + '%';
  }
})
```

```

    this.InvestmentListItem = InvestmentListItem;
  }) (React);

```

As you can see, it was as simple as nesting another button component and passing down the `onClickDelete` prop value as its `onClick` prop.

React normalizes events so that they have consistent properties across different browsers, but its naming conventions and syntax is similar to inline JavaScript code in HTML. To get a comprehensive list of the supported events, you can check the official documentation at <http://facebook.github.io/react/docs/events.html>.

Component state

Until now, we've dealt with React as a stateless rendering engine, but as we know, applications have state, especially when using forms. So, how would we implement the `NewInvestment` component in order for it to keep hold of the values of the investment being created and then notify an observer once the user completed the form?

To help us implement this behavior, we are going to use another component internal API—its **state**.

Let's take the following acceptance criterion:

Given that the inputs of the `NewInvestment` component are correctly filled, when the form is submitted, it should notify the `onCreate` observer with the investment attributes:

```

describe("NewInvestment", function() {
  var TestUtils = React.addons.TestUtils;
  var component, onCreateSpy;

  function findNodeWithClass (className) {
    return TestUtils.findRenderedDOMComponentWithClass(component,
      className).getDOMNode();
  }

  beforeEach(function() {
    onCreateSpy = jasmine.createSpy('onCreateSpy');
    component = TestUtils.renderIntoDocument(
      <NewInvestment onCreate={onCreateSpy}/>
    );
  });
});

```



```
describe("with its inputs correctly filled", function() {
  beforeEach(function() {
    var stockSymbol = findNodeWithClass('new-investment-stock-
symbol');
    var shares = findNodeWithClass('new-investment-shares');
    var sharePrice = findNodeWithClass('new-investment-share-
price');

    TestUtils.Simulate.change(stockSymbol, { target: { value: 'AOUE'
}});
    TestUtils.Simulate.change(shares, { target: { value: '100' }});
    TestUtils.Simulate.change(sharePrice, { target: { value: '20'
}});
  });

  describe("when its form is submitted", function() {
    beforeEach(function() {
      var form = component.getDOMNode();
      TestUtils.Simulate.submit(form);
    });

    it("should invoke the 'onCreate' callback with the investment
attributes", function() {
      var investmentAttributes = { stockSymbol: 'AOUE', shares:
'100', sharePrice: '20' };

      expect(onCreateSpy).toHaveBeenCalledWith(investmentAttribut
es);
    });
  });
});
```

This spec is basically using every trick we've learned until now, so without getting into the details, let's dive directly into the component implementation.

The first thing that any component with state must declare is its initial state by defining a `getInitialState` method, as follows:

```
var NewInvestment = React.createClass({
  getInitialState: function () {
    return {
      stockSymbol: '',
      shares: 0,
      sharePrice: 0
    };
  }
});
```

```

    },

    render: function () {
      var state = this.state;

      return <form className="new-investment">
        <h1>New investment</h1>
        <label>
          Symbol:
          <input type="text" ref="stockSymbol" className="new-
investment-stock-symbol" value={state.stockSymbol} maxLength="4"/>
        </label>
        <label>
          Shares:
          <input type="number" className="new-investment-shares"
value={state.shares}/>
        </label>
        <label>
          Share price:
          <input type="number" className="new-investment-share-price"
value={state.sharePrice}/>
        </label>
        <input type="submit" className="new-investment-submit"
value="Add"/>
      </form>;
    }
  });

```

As illustrated in the preceding code, we are clearly defining the initial state of our form, and at the render method, we pass the state as `value` props to the input components.

If you run this example in a browser, you will notice that you won't be able to change the values of the inputs. You can focus on the inputs, but trying to type won't change its values, and that is because of the way React works.

Unlike HTML, React components must represent the state of the view at any point in time and not only at initialization time. If we want to change the value of an input, we need to listen for the `onChange` events of the inputs and, with that information, update the state. The change in the state will then trigger a render that will update the value on screen.

To demonstrate how this works, let's implement this behavior at the `stockSymbol` input.

First, we need to change the render method, adding a handler to the onChange event:

```
<input type="text" ref="stockSymbol" className="new-investment-stock-symbol" value={state.stockSymbol} maxLength="4" onChange={this._handleStockSymbolChange}/>
```

Once the event is triggered, it will invoke the `_handleStockSymbolChange` method. Its implementation should update the state by invoking the `this.setState` method with the new value of the input, as follows:

```
var NewInvestment = React.createClass({
  getInitialState: function () {
    // ... Method implementation
  },

  render: function () {
    // ... Method implementation
  },

  _handleStockSymbolChange: function (event) {
    this.setState({ stockSymbol: event.target.value });
  }
});
```

The event handler is a good place to perform simple validation or transformation in the input data before passing it to the state.

As you can see, this is a lot of boilerplate code just to handle a single input. Since we are not implementing any custom behavior into our event handlers, we can use a special React feature that implements this "linked state" for us.

We are going to use a **Mixin** called `LinkStateMixin`; but first, what is a Mixin? It is a way to share common functionality between components, which, in this case, is the "linked state". Take a look at the following code:

```
var NewInvestment = React.createClass({
  mixins: [React.addons.LinkStateMixin],

  // ...

  render: function () {
    // ...
    <input type="text" ref="stockSymbol" className="new-investment-stock-symbol" valueLink={this.linkState('stockSymbol')} maxLength="4" />
    // ...
  }
});
```

```
    }
  });
```

`LinkStateMixin` works by adding the `linkState` function to the component, and instead of setting the value of the input, we set a special prop called `valueLink` with the link object returned by the function `this.linkState`.

The `linkState` function expects the name of the attribute of the **state** that it should link to the value of the input.

Component life cycle

As you might have noticed, React has an opinionated view on a component's API. But it also has a very strong opinion on its life cycle, allowing us developers to add hooks to create custom behavior and perform cleanup tasks as we develop our components.

This is one of React's greatest triumphs because it is through this standardization that we can create bigger and better components by composition; through that, we can use not only our components, but other people's components.

To demonstrate one use case, we are going to implement a very simple behavior: on page load, we want the new investment form stock symbol input to be focused so that a user can start typing right away.

But, before we can start writing the test, there is just one thing that we will need to do. As mentioned earlier, `TestUtils.renderIntoDocument` doesn't actually render anything in the document, but instead on a detached node. So, if we use it to render our component, we won't be able to make the assertion regarding the input's focus.

So, yet again, we have to use the `setFixtures` method to actually render the React component in the document, as follows:

```
/**
 * Uses jasmine-jquery fixtures to actually render in the document.
 * React.TestUtils.renderIntoDocument renders in a detached node.
 *
 * This was required to test the focus behavior.
 */
function actuallyRender (component) {
  setFixtures('<div id="application-container"></div>');
  var container = document.getElementById('application-container');
  return React.render(component, container);
}
```

```
describe("NewInvestment", function() {
  var TestUtils = React.addons.TestUtils;
  var component, stockSymbol;

  function findNodeWithClass (className) {
    return TestUtils.findRenderedDOMComponentWithClass(component,
      className).getDOMNode();
  }

  beforeEach(function() {
    component = actuallyRender(<NewInvestment
      onCreate={onCreateSpy}/>);
    stockSymbol = findNodeWithClass('new-investment-stock-symbol');
  });

  it("should have its stock symbol input on focus", function() {
    expect(stockSymbol).toBeFocused();
  });
});
```

With this small change completed, and the spec written, we can get back to the implementation.

React gives a few hooks that we can implement custom code in our component during its life cycle; they are as follows:

- `componentWillMount`
- `componentDidMount`
- `componentWillReceiveProps`
- `shouldComponentUpdate`
- `componentWillUpdate`
- `componentDidUpdate`
- `componentWillUnmount`

To implement our custom behavior, we are going to use the `componentDidMount` hook that is called only once, right after the component was rendered and attached into a DOM element.

So, what we want to do is somehow inside this hook, get access to the input DOM element and trigger its focus. We already know how to get a DOM node; it is through the `getDOMNode` API. But, how do we get the input's React element?

React's other feature for this problem is called **ref**. It is basically a way to give names to a component's children to allow later access.

Since we want the stock symbol input, we need to add a `ref` attribute to it, as follows:

```
<input type="text" ref="stockSymbol" className="new-investment-stock-symbol" valueLink={this.linkState('stockSymbol')} maxLength="4" />
```

Then, at the `componentDidMount` hook, we can get the input by its `ref` name and then its DOM element and trigger the focus, as follows:

```
var NewInvestment = React.createClass({
  // ...
  componentDidMount: function () {
    this.refs.stockSymbol.getDOMNode().focus();
  },
  // ...
});
```

The other hooks are setup in the same way, by simply defining them on the class definition object as properties. But each is called on different occasions, and has different rules. The official documentation is a great resource on their definition and possible use-cases which can be found at <http://facebook.github.io/react/docs/component-specs.html#lifecycle-methods>.

Composing components

We've talked a lot about **composability** in the way of creating components by composing React's default components. However, we haven't showed how to compose custom components into bigger components.

As you might have guessed, this should be a pretty simple exercise, and to demonstrate how this works, we are going to implement a component to list investments, as follows:

```
var InvestmentList = React.createClass({
  render: function () {
    var onClickDelete = this.props.onClickDelete;

    var listItems = this.props.investments.map(function (investment) {
      return <InvestmentListItem investment={investment}
        onClickDelete={onClickDelete.bind(null,
investment)} />;
    });
  }
});
```

```
    });  
  
    return <ul className="investment-list">{listItems}</ul>;  
  }  
});
```

It is as simple as using the already available `InvestmentListItem` global variable as the root element of the `InvestmentList` component.

The component expects an `investments` prop to be an array of investments. It then maps it through creating an `InvestmentListItem` element for each investment in the array.

Finally, it uses the `listItems` array as the children of an `ul` element, effectively defining how to render the list of investments.

Summary

React is a rapidly evolving library that is getting a lot of traction by the JavaScript community; it introduced some interesting patterns and questioned some well-established dogmas as it is continually improving the development of rich web applications.

This chapter's goal was not to give an in-depth look into the library but an overview of its primary features and philosophies. It demonstrated that it is possible to do test-driven development while coding your interface with React.

You learned about **prop** and **state** and their differences: A **prop** is not owned by the component, and should, if needed, be changed by its parent. The **state** is the data that the component owns. It can be changed by the component, and by doing so, a new render is triggered.

The fewer components with state you have in your application, the easier it is going to be to reason about it and test it.

It is through React's opinionated API and life cycle that we can get the maximum benefit of composability and code reuse.

As you move into application development with React, it is recommended that you learn about Flux, the recommended architecture to build applications by Facebook, at <http://facebook.github.io/flux/>.

8

Build Automation

We saw how to create an application from the ground up using tests with Jasmine. However, as the application grows and the number 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 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 Stock is available once Investment is loaded. Here's how we do it:

```
<script type="text/javascript" src="src/Stock.js"></"script>
<script type="text/javascript" src="src/Investment.js"></"script>
```

However, that can soon become cumbersome and unmanageable.

Another problem is the number of requests the application uses to load all of its files; it can get up to hundreds once the application starts to grow.

So, we have a paradox here; although it is good to break it up into small modules for code maintainability, 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 modules

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 **webpack**.

Module bundler – webpack

Webpack is a module bundler created by Tobias Koppers to help create big and modular frontend JavaScript applications.

Its main difference from other solutions is its support for any type of module system (AMD and CommonJS), languages (CoffeeScript, TypeScript, and JSX) and even assets (images and templates) through loaders.

You read it right, even images; if in a React application, everything is a component, in a webpack project, everything is a module.

It builds a dependency graph of all your assets, serving them in a development environment and optimizing them for production.

Module definition

JavaScript is a language based on the ECMA Script specification that, until version 6, still didn't have a standard definition of a module. This lack of formal standards led to a number of competing community standards (AMD and CommonJS) and implementations (RequireJS and browserify).

Now, there is a standard to follow, but unfortunately there is no support for it in modern browsers, so which style should we use to write our modules?

The good news is that it is possible to use ES6 today through transpilers, which gives us a future-proof advantage.

A popular transpiler is **Babel** (<http://babeljs.io/>), which we are going to use with webpack through a loader.

We'll see how to use it with webpack in a moment, but first it is important to understand what makes an ES6 module. Here is a simple definition without any dependency:

```
function MyModule () {};  
export default MyModule;
```

Let's compare it to the way we've been declaring modules until now. The next example shows how that code would be if written using the conventions presented in *Chapter 3, Testing Frontend Code*:

```
(function () {  
  function MyModule() {};  
  this.MyModule = MyModule;  
})();
```

The biggest difference is the lack of an IIFE. An ES6 module, by default, has a scope of its own, so it is impossible to pollute the global namespace by accident.

The second difference is that the module value is no longer being attached to the global object, but instead being exported as the default module value:

```
function MyModule () {};  
export default MyModule;
```

Regarding a module's dependencies, up until now, everything was globally available, so we passed the dependencies to the module as parameters to the IIFE, as shown here:

```
(function ($) {  
  function MyModule() {};  
  this.MyModule = MyModule;  
})(jQuery);
```

However, as you start using ES6 modules on the project, there will be no more global variables. So, how do you get those dependencies into the module?

If you remember from before, the ES6 example was exporting the module value through the `export default` syntax. So, given a module has a value, all we have to do is ask for it as a dependency. Let's add the jQuery dependency to our ES6 module:

```
import $ from 'jQuery';  
function MyModule () {};  
export default MyModule;
```

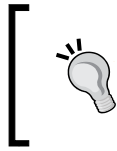
Here, `$` represents the name of the variable the dependency will be loaded into, and `jQuery` is the filename.

It is also possible to export multiple values as the result of a module and import these values into different variables, but for the scope of this book, default values will suffice.

The ES6 standard introduces a number of different constructs to the JavaScript language that are also beyond the scope of this book. For more information, check Babel's excellent documentation at <http://babeljs.io/docs/learn-es6/>.

Webpack project setup

Webpack is available as an NPM package, and its setup is very simple as it is going to be demonstrated in the next sections.



It is important to understand the difference between NPM and Node.js. NPM is both a package manager and a package format, while Node.js is a platform that NPM modules usually run.

Managing dependencies with NPM

We already got an embryo of a Node.js project, but as we are going to start using more dependencies throughout this chapter, we are going to need a formal definition of all the NPM packages that the project depends on.

To define the project as an NPM package, and at the same time all of its dependencies, we need to create a special file called `package.json` at the root folder of the application. It can be easily created through a single command:

```
npm init
```

It will prompt for a number of questions about the project that can all be left with their default values. In the end, you should have a file with content similar to the following output depending on your folder name:

```
{
  "name": "jasmine-testing-project",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
Is this ok? (Yes)
```

The next step is to install all of our dependencies, which, at the moment, is only `express`.

```
npm install --save express
```

The previous command will not only install `express` as described in *Chapter 4, Asynchronous Testing – AJAX*, but will also add it as a dependency to the `package.json` file. On running the `npm init` command as done previously, we get the following output showing the `dependencies` attribute:

```
{
  "name": "jasmine-testing-project",
  "version": "1.0.0",
  "dependencies": {
    "express": "*"
  }
}
```

```

    "description": "",
    "main": "index.js",
    "scripts": {
      "test": "echo \"Error: no test specified\" && exit 1"
    },
    "author": "",
    "license": "ISC",
    "dependencies": {
      "express": "^4.12.0"
    }
  }
}

```

Now that we understand how to manage the dependencies of our project, we can install **webpack** and **Babel** as development dependencies to start bundling our modules, as follows:

```
npm install --save-dev babel-loader webpack webpack-dev-server
```

The final step is to add a script in `package.json` to start the development server:

```

"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1",
  "dev": "webpack-dev-server"
}

```

This allows us to start the development server with a simple command:

```
npm run dev
```

The actual location of the `webpack-dev-server` executable is in the `./node_modules/.bin` folder. So, `npm run dev` is the same as:

```
./node_modules/.bin/webpack-dev-server
```

It works because when you run `npm run <scriptName>`, NPM adds the `./node_modules/.bin` folder to the path.

Webpack configuration

Next, we need to configure webpack so that it knows what files to bundle. This can be achieved by creating a `webpack.config.js` file at the root folder of the project. Its content should be:

```

module.exports = {
  context: __dirname,
  entry: {
    spec: [

```

```
    './spec/StockSpec.js',
    './spec/InvestmentSpec.js',
    './spec/components/NewInvestmentSpec.jsx',
    './spec/components/InvestmentListItemSpec.jsx',
    './spec/components/InvestmentListSpec.jsx'
  ],
},

output: {
  filename: '[name].js'
},

module: {
  loaders: [
    {
      test: /\.js$|\.jsx$/,
      exclude: /node_modules/,
      loader: 'babel-loader'
    }
  ]
}
};
```

There are a few key points about this configuration file:

- The `context` directive tells webpack to look for modules in `__dirname`, meaning the project's root folder.
- The `entry` directive specifies the application's entry points. Since we are only doing testing at the moment, there is a single entry point named `spec` that refers to all of our spec files.
- The `output.filename` directive is here to inform the filename of each of the entry points. The `[name]` pattern will be replaced by an entry point name on compilation. So `spec.js` will actually contain all of our spec code.
- The `module.loaders` final directive tells webpack how to deal with different file types. We are using the `babel-loader` parameter here to add support for ES6 modules and the JSX syntax on our source files. The `exclude` directive is important so as not to waste compiling any dependency from the `node_modules` folder.

With this setup completed, you can start the development server and check what the transpiled bundle looks like at <http://localhost:8080/spec.js> (the filename defined in the configuration file).

At this point, the webpack configuration is complete, and we can move to adapt the Jasmine runner to run the specs.

The spec runner

As stated previously, we are using webpack to compile and bundle the source files, so the Jasmine spec is about to become a lot simpler:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>Jasmine Spec Runner v2.1.3</title>

  <link rel="shortcut icon" type="image/png" href="lib/jasmine-2.1.3/
jasmine_favicon.png">
  <link rel="stylesheet" href="lib/jasmine-2.1.3/jasmine.css">

  <script src="lib/jasmine-2.1.3/jasmine.js"></script>
  <script src="lib/jasmine-2.1.3/jasmine-html.js"></script>
  <script src="lib/jasmine-2.1.3/boot.js"></script>

  <script src="lib/jquery.js"></script>
  <script src="lib/jasmine-jquery.js"></script>

  <script src="lib/mock-ajax.js"></script>

  <script src="spec/SpecHelper.js"></script>

  <script src="spec.js"></script>
</head>
<body>
</body>
</html>
```

There are a few takeaways:

First, we no longer need the JSX transformer hack explained in the previous chapter; the transformation is now done by webpack and the babel-loader. As a result, we can use the default Jasmine boot just fine.

Second, we've chosen to leave the test runner dependencies as global (Jasmine, Mock Ajax, Jasmine JQuery, and the Spec helper). Leaving them global makes things a lot simpler for our test runner, and we don't hurt our code as far as modularity is concerned.

At this moment, trying to run the tests at `http://localhost:8080/SpecRunner.html` should produce a lot of fails due to missing references. That is because we still need to convert our specs and sources into ES6 modules.

Testing a module

To be able to run all the tests requires that all the source and spec files be converted into ES6 modules. At the specs, it means adding, as dependencies all the source modules:

```
import Stock from '../src/Stock';
describe("Stock", function() {
  // the original spec code
});
```

At the source files, it means declaring all the dependencies as well as exporting its default value, as follows:

```
import React from 'react';
var InvestmentListItem = React.createClass({
  // original code
});
export default InvestmentListItem;
```

Once all the code is converted, the tests should work upon starting the development server and pointing the browser once again to the runner URL.

Test runner: Karma

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 that powers the Safari browser) and **Karma**, a test runner.

The setup is very simple; using NPM, we once again install some dependencies:

```
npm install --save-dev karma karma-jasmine karma-webpack karma-phantomjs-launcher es5-shim
```

The only strange dependency here is the `es5-shim`, which is used to give PhantomJS support for some ES5 features that it still is missing, and React requires.

The next step is creating a configuration file, named `karma.conf.js`, for Karma at the project's root folder:

```
module.exports = function(config) {
  config.set({
    basePath: '.',

    frameworks: ['jasmine'],
```

```

    browsers: ['PhantomJS'],

    files: [
        // shim to workaroud PhantomJS 1.x lack of 'bind' support
        // see: https://github.com/ariya/phantomjs/issues/10522
        'node_modules/es5-shim/es5-shim.js',
        'lib/jquery.js',
        'lib/jasmine-jquery.js',
        'lib/mock-ajax.js',
        'spec/SpecHelper.js',
        'spec/**/*.Spec.*'
    ],

    preprocessors: {
        'spec/**/*.Spec.*': ['webpack']
    },

    webpack: require('./webpack.config.js'),
    webpackServer: { noInfo: true },
    singleRun: true
  });
};

```

In it, we set up the Jasmine framework and the PhantomJS browser:

```

    frameworks: ['jasmine'],
    browsers: ['PhantomJS'],

```

Fix the browser compatibility issues on PhantomJS by loading `es5-shim`, as shown in the following code:

```

    // shim to workaroud PhantomJS 1.x lack of 'bind' support
    // see: https://github.com/ariya/phantomjs/issues/10522
    'node_modules/es5-shim/es5-shim.js',

```

Load the test runner dependencies, which were previously global in the `SpecRunner.html` file, as shown in the following code:

```

    'lib/jquery.js',
    'lib/jasmine-jquery.js',
    'lib/mock-ajax.js',
    'spec/SpecHelper.js',

```

Finally, load all the specs, as follows:

```

    'spec/**/*.Spec.*',

```


By now, you can remove the `SpecRunner.html` file, the `spec` entry in the `webpack.config.js` file, and the `lib/jasmine-2.1.3` folder.

Run the tests by invoking Karma, which will print the test results in the console, as follows:

```
./node_modules/karma/bin/karma start karma.conf.js
> investment-tracker@0.0.1 test /Users/paulo/Dropbox/jasmine_book/second_
edition/book/chapter_8/code/webpack-karma
> ./node_modules/karma/bin/karma start karma.conf.js
INFO [karma]: Karma v0.12.31 server started at http://localhost:9876/
INFO [launcher]: Starting browser PhantomJS
INFO [PhantomJS 1.9.8 (Mac OS X)]: Connected on socket
cGbcpaDgXl4wdyzLZh with id 37309028
PhantomJS 1.9.8 (Mac OS X): Executed 36 of 36 SUCCESS (0.21 secs / 0.247
secs)
```

To make it simpler to run the tests, it is possible to change the `package.json` project file and describe its test script:

```
"scripts": {
  "test": "./node_modules/karma/bin/karma start karma.conf.js",
  "dev": "webpack-dev-server"
},
```

You can then run the tests by simply invoking the following:

```
npm test
```

Quick feedback loop

Automated testing is all about the quick feedback loop, so imagine being able to have the tests running in the console and the application refreshing on the browser after any file change. Would that be possible? The answer is yes!

Watch and run the tests

Via a simple parameter while starting Karma, we can achieve testing nirvana, as follows:

```
./node_modules/karma/bin/karma start karma.conf.js --auto-watch --no-
single-run
```

Try it by yourself; run this command, change a file, and see the tests running automatically – like magic.

Once again, we don't want to remember these complicated commands, so let's add another script to the `package.json` file:

```
"scripts": {  
  "test": "./node_modules/karma/bin/karma start karma.conf.js",  
  "watch-test": "./node_modules/karma/bin/karma start karma.conf.js  
    --auto-watch --no-single-run",  
  "dev": "webpack-dev-server"  
},
```

We can run it through the following command:

```
npm run watch-test
```

Watch and update the browser

To achieve development nirvana, we are also just a parameter away.

While starting the development server, add the following to the `package.json` file:

```
./node_modules/.bin/webpack-dev-server --inline -hot
```

Once again, try it on your browser; change a file in the text editor and the browser should refresh.

You are also encouraged to update the `package.json` file with these new parameters so that running `npm run dev` gets you the goodness of "live reload".

Optimizing for production

The final step of our module bundler goal is to generate a minified and ready-for-production file.

Most of the configuration is complete, missing just a few more steps.

The first step is to set up an entry point for the application, then an index file that will start it all, `index.js`, is to be placed inside the `src` folder with the contents as follows:

```
import React from 'react';  
import Application from './Application.jsx';  
  
var mountNode = document.getElementById('application-container');  
React.render(React.createElement(Application, {}), mountNode);
```

We haven't covered in detail the implementation of this file in the book, so be sure to check the attached source files to understand better how it works.

In the webpack configuration file, we need to add both an output path to indicate where the bundled files will be placed and the new entry file we just created, as follows:

```
module.exports = {
  context: __dirname,
  entry: {
    index: './src/index.js'
  },

  output: {
    path: 'dist',
    filename: '[name] - [hash].js'
  },

  module: {
    loaders: [
      {
        test: /\.js$|\.jsx$/,
        exclude: /node_modules/,
        loader: 'babel-loader'
      }
    ]
  }
};
```

Then, all that is left is to create a build task in our `package.json` file:

```
"scripts": {
  "test": "./node_modules/karma/bin/karma start karma.conf.js",
  "watch-test": "./node_modules/karma/bin/karma start karma.conf.js --auto-watch --no-single-run",
  "build": "webpack -p",
  "dev": "webpack-dev-server --inline --hot"
},
```

Run it and check the built files into the `dist` folder, as follows:

npm run build

Static code analysis: JSHint

As stated in the first chapter, JavaScript is not a compiled language, but running the code (as in the case of automated testing) is not the only way to check for errors.

A whole class of tools is able to read source files, interpret them, and look for common errors or bad practices without needing to actually run the source files.

A very popular tool is **JSHint**—a simple binary that can also be installed through NPM, as follows:

```
npm install --save-dev jshint jsxhint
```

You can see that we are also installing **JSXHint**, another tool to perform static analysis of JSX files. It is basically a wrapper around the original JSHint while performing the JSX transformations.

If you remember from the previous chapter, JSXTransformer doesn't change the line numbers, so a warning in a given line number on a JavaScript file will be in the same line number in the original JSX file.

To execute them is very simple, as follows:

```
./node_modules/.bin/jshint .
./node_modules/.bin/jsxhint .
```

However, it is also a good idea to have them running whenever we run the tests:

```
"scripts": {
  "start": "node bin/server.js",
  "test": "./node_modules/.bin/jshint . && ./node_modules/.bin/
jsxhint . && ./node_modules/karma/bin/karma start karma.conf.js",
  "watch-test": "./node_modules/karma/bin/karma start karma.conf.js
--auto-watch --no-single-run",
  "build": "webpack -p",
  "dev": "webpack-dev-server --inline --hot"
},
```

The final step is configuring what errors we want JSHint and JSXHint to catch. Once again, we create another configuration file at the root folder of our project, this time called `.jshintrc`:

```
{
  "esnext": true,
  "undef": true,
  "unused": true,
  "indent": 2,
  "noempty": true,
```

```
"browser": true,
"node": true,
"globals": {
  "jasmine": false,
  "spyOn": false,
  "describe": false,
  "beforeEach": false,
  "afterEach": false,
  "expect": false,
  "it": false,
  "xit": false,
  "setFixtures": false
}
}
```

This is a list of option flags either being enabled or disabled, where the most important are the following:

- `esnext`: This flag tells us we are using the ES6 version
- `unused`: This flag breaks on any unused declared variable
- `undef`: This option flag breaks on any variable being used without being declared

There is also a list of `globals` variables used by the tests to prevent errors due to the `undef` flag.

Head over to the JSHint website at <http://jshint.com/docs/options/> for a complete list of options.

The only missing step is preventing the linter from running in other people's code (Jasmine, React, and so on). This is possible by simply creating a file with the folders it should ignore. This file called `.jshintignore` should contain:

- `node_modules`
- `lib`

To run the static analysis and all the tests is now as simple as this:

```
npm test
```

Continuous integration – Travis-CI

We have created a great deal of automation around the project, which is great for onboarding a new developer on the team; running the test is just two commands away:

```
npm install
npm test
```

However, that is not the only advantage; we can have the tests running via these same two commands while on a continuous integration environment.

To demonstrate a possible setup, we are going to use Travis-CI (<https://travis-ci.org>), a free solution for open source projects.

Before we can start, it is required that you have a GitHub (<https://github.com/>) account and that the project is already hosted there. I expect that you are already familiar with git (<http://www.git-scm.com/>) and GitHub.

Once you are ready, we can start the Travis-CI setup.

Adding a project to Travis-CI

Before we can add Travis-CI support to the project, first we need to add the project to Travis-CI.

Go to the Travis-CI website at <https://travis-ci.org> and click on **Sign in with GitHub** in the top-right corner.

Enter your GitHub credentials and once you have signed in, it should show you the list with all your repositories:

If your repository doesn't show up, you can click on the **Sync Now** button to make Travis-CI update the list.

Once your repository appears, enable it by clicking on the switch. This will set up hooks on your GitHub project, so Travis-CI gets notified on any change pushed to the repository.

Project setup

Setting up a Travis-CI project couldn't be simpler. Since we have our build process and tests all scripted, all we have to do is tell Travis-CI what runtime it should use.

Travis-CI knows that Node.js project dependencies are installed through `npm install` and that tests are run through `npm test`, so there is no extra step to get our tests running.

Create a new file at the project root directory called `.travis.yml` and configure the language for Travis as Node.js:

```
language: node_js
```

And that is all there is to it.

The steps taken to use Travis-CI were pretty straightforward, and it should be pretty simple to apply these same concepts to other continuous integration environments, such as Jenkins (<http://jenkins-ci.org/>).

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 learned about webpack and how it can be used to manage the dependencies between your modules and help you generate the production code (packed and minified).

The power of static code analysis in helping us find bugs even before the code runs.

You have also seen how to run your specs headlessly, and even automatically, letting you focus on the code editor all the time.

Finally, we have seen how simple it is to use a continuous integration environment and how we can use this powerful concept to keep our projects always tested.

Index

A

- acceptance criterion** 53, 54
- assertion** 10
- asynchronous specs**
 - about 58, 59
 - timeout 59
- attributes, React component** 80-82

B

- Babel**
 - about 97
 - URL 94
- basic View coding rules**
 - about 40
 - DOM element, encapsulating 41-43
- behavior checking** 61, 62
- behavior-driven development (BDD)** 3
- built-in matchers**
 - toBe 27
 - toBeCloseTo 31
 - toBeDefined 30
 - toBeFalsy 28, 29
 - toBeGreaterThan 31
 - toBeLessThan 31
 - toBeNaN 29
 - toBeNull 29
 - toBeTruthy 28, 29
 - toBeUndefined 29
 - toContain 30
 - toEqual 26
 - toMatch 31
 - toThrow 32

C

- command-line interface (CLI)** 39
- composability, React component** 91, 92
- continuous integration**
 - Travis-CI 107
- custom matchers**
 - about 21-25
 - reference link 26

D

- describe function** 9
- DOM events**
 - testing 63, 64
- doubles** 61

E

- Envjs** 5
- equals operator (==)** 27
- events, React component**
 - about 82-85
 - reference link 85
- Express** 56

F

- fakes** 65
- fake XMLHttpRequest** 67-69

G

- git**
 - URL 107
- GitHub**
 - URL 107

H

headless browser 5

HTML fixtures

functions 38

using 36-40

I

immediately invoked function expression (IIFE) 35

Investment Tracker application

about 7, 8

mockup interface 34

setup function, implementing 14-17

teardown function, implementing 14-17

J

Jasmine

about 3

basics 8-13

downloading 4, 5

JSX, using with 78, 79

URL 4

Jasmine Ajax

about 67

adding, to project 67

fake XMLHttpRequest 67-69

URL 67

jasmine-jquery.js

URL, for downloading 33

Jasmine stubs 65, 66

JavaScript

about 94

limitations 1, 2

Jenkins

URL 108

jQuery 33

jQuery issue, GitHub ticket

reference link 39

jquery.js

URL, for downloading 33

jQuery matchers

toBeDisabled 51

toBeFocused 51

toBeMatchedBy 49

toContainElement 50

toContainHtml 50

toHaveAttr 50

toHaveValue 50

Views, testing with 48, 49

JSHint

about 105

URL 106

JSX

about 76, 78

URL 78

using, with Jasmine 78, 79

JSXHint 105

K

Karma 100-102

L

life cycle, React component 89-91

M

matchers

about 10, 20

built-in matchers 26

custom matchers 21-25

reference link 51

mockup interface, Investment Tracker application 34

Model View Controller (MVC) 1, 34

module bundler

webpack 94

module pattern 35, 36

N

nested describes 18

Node.js

about 5

installing 55

O

Object.seal function 2

object's functions

spying on 62, 63

observers

Views, integrating with 43-48

P

PhantomJS 5, 100

production

optimizing for 103, 104

project

adding, to Travis-CI 107

project setup, React component 72

props 75, 80

Q

quick feedback loop

about 102

browser, updating 103

browser, watching 103

tests, running 102

tests, watching 102

R

React component

about 71-75

attributes 80-82

composability 91, 92

events 82-85

life cycle 89-91

project setup 72

state 85-89

React Starter Kit Version 0.12.2

URL, for downloading 72

ref 91

return on investment (ROI) 18

runner 4

S

same origin policy (SOP) 39

scenario, Node.js

server, coding 55, 56

server, running 56

setting up 55

setup function

using 19

shared behavior

spec, coding with 19, 20

spec

about 4

asynchronous setups 57, 58

asynchronous teardowns 57, 58

coding, with shared behavior 19, 20

writing 57

SpecRunner.html file

spec runner, webpack 99

spy 62

state, React component 85-89

static code analysis

JSHint 105, 106

strict equals operator (===) 27

stubs 65

T

teardown function

using 19

test double 61

test-driven development (TDD) 3

test runner

Karma 100-102

test unit 3

toBe built-in matcher 27

toBeCloseTo built-in matcher 31

toBeDefined built-in matcher 30

toBeDisabled jQuery matcher 51

toBeFalsy matcher 28, 29

toBeFocused jQuery matcher 51

toBeGreaterThan built-in matcher 31

toBeLessThan built-in matcher 31

toBeMatchedBy jQuery matcher 49

toBeNaN built-in matcher 29

toBeNull built-in matcher 29

toBeTruthy matcher 28, 29

toBeUndefined built-in matcher 29

toContain built-in matcher 30

toContainElement jQuery matcher 50

toContainHtml jQuery matcher 50

toEqual built-in matcher 26

toHaveAttr jQuery matcher 50

toHaveValue jQuery matcher 50

toMatch built-in matcher 31

toThrow built-in matcher 32

Travis-CI

project, adding to 107

project, setting up 108

URL 107

U

user stories 3

V

Views

about 34

integrating, with observers 43–48

testing, with jQuery matchers 48, 49

Virtual DOM 75, 76

W

webpack

about 94, 97

configuring 97, 98

module definition 94, 95

module, testing 100

spec runner 99

webpack project setup

about 95

dependencies, managing with NPM 96, 97



Thank you for buying **Jasmine JavaScript Testing** *Second Edition*

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 that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at www.packtpub.com.

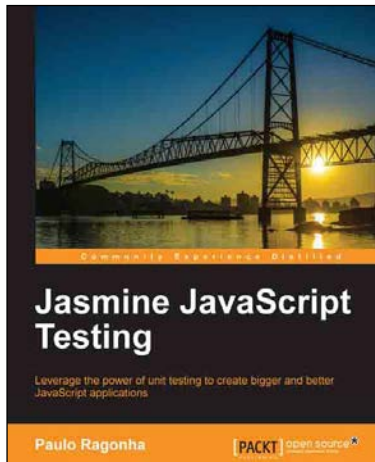
About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around open source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each open source project about whose software a book is sold.

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, then please 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.



Jasmine JavaScript Testing

ISBN: 978-1-78216-720-4

Paperback: 146 pages

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

1. Learn the power of test-driven development while creating a fully-featured web application.
2. Understand the best practices for modularization and code organization while putting your application to scale.
3. Leverage the power of frameworks such as BackboneJS and jQuery while maintaining the code quality.



Web App Testing Using Knockout.JS

ISBN: 978-1-78398-284-4

Paperback: 154 pages

Design, implement, and maintain a fully tested JavaScript web application using Knockout.JS

1. Test JavaScript web applications using one of the most known unit testing libraries – Jasmine.js.
2. Leverage the two way bindings and dependency tracking mechanism to test web applications using Knockout.js.
3. The book covers different JavaScript application testing strategies supported by real-world examples.

Please check www.PacktPub.com for information on our titles

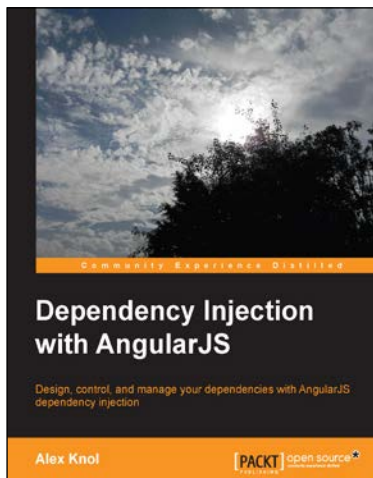


JavaScript Unit Testing

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.



Dependency Injection with AngularJS

ISBN: 978-1-78216-656-6 Paperback: 78 pages

Design, control, and manage your dependencies with AngularJS dependency injection

1. Understand the concept of dependency injection.
2. Isolate units of code during testing JavaScript using Jasmine.
3. Create reusable components in AngularJS.

Please check www.PacktPub.com for information on our titles