



Community Experience Distilled

Learning Behavior-driven Development with JavaScript

Create powerful yet simple-to-code BDD test suites in JavaScript using the most popular tools in the community

Enrique Amodeo

[PACKT] open source*
community experience distilled
PUBLISHING

Learning Behavior-driven Development with JavaScript

Create powerful yet simple-to-code BDD test suites in JavaScript using the most popular tools in the community

Enrique Amodeo



BIRMINGHAM - MUMBAI

Learning Behavior-driven Development with JavaScript

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: February 2015

Production reference: 1130215

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

ISBN 978-1-78439-264-2

www.packtpub.com

Credits

Author	Project Coordinator
Enrique Amodeo	Judie Jose
Reviewers	Proofreaders
Domenico Luciani	Stephen Copestake
Mihir Mone	Maria Gould
Takeharu Oshida	Paul Hindle
Juri Strumpflohner	
Commissioning Editor	Indexer
Pramila Balan	Priya Sane
Acquisition Editor	Graphics
Richard Brookes-Bland	Sheetal Aute
Content Development Editors	Production Coordinator
Sriram Neelakantan	Nitesh Thakur
Sharvari Tawde	Cover Work
	Nitesh Thakur
Technical Editor	
Indrajit A. Das	
Copy Editors	
Karuna Narayanan	
Laxmi Subramanian	

About the Author

Enrique Amodeo is an experienced software engineer currently working and living in Berlin. He is a very eclectic professional with very different interests and more than 15 years of experience. Some of his areas of expertise are JS, BDD/TDD, REST, NoSQL, object-oriented programming, and functional programming.

As an agile practitioner, he uses BDD and emergent design in his everyday work and tries to travel light. Experienced in both frontend and server-side development, he has worked with several technical stacks, including Java/JEE, but since 2005, he prefers to focus on JS and HTML5. He is now very happy to be able to apply his JS knowledge to the server-side development, thanks to projects such as Node.js.

He also has written a book in Spanish on designing web APIs, following the REST and hypermedia approach (https://leanpub.com/introduccion_apis_rest).

I would like to thank my wife for making this book possible.
She is the one who supported me and reminded me to "continue
writing that difficult chapter" whenever I started thinking of
doing something else. Without her, I would probably have
never completed this book!

About the Reviewers

Domenico Luciani is a software and web developer and compulsive coder. He is curious and is addicted to coffee. He is a computer science student and a passionate pentester and computer-vision fanatic.

Having fallen in love with his job, he lives in Italy; currently, he is working for many companies in his country as a software/web developer. You can find more information on him at <http://dlion.it/>.

Mihir Mone is a postgraduate from Monash University, Australia. Although he did his post graduation in network computing, these days, he mainly does web and mobile development.

After spending some time fiddling around with routers and switches, he quickly decided to build upon his passion for web development—not design, but development. Building web systems and applications rather than websites with all their fancy Flash animations was something that was very interesting and alluring to him. He even returned to his alma mater to teach web development in order to give back what he had learned.

These days, he works for a small software/engineering house in Melbourne, doing web development and prototyping exciting, new ideas in the data visualization and UX domains.

He is also a big JavaScript fan and has previously reviewed a few books on jQuery and JavaScript. He is a Linux enthusiast and a big proponent of the OSS movement. He believes that software should always be free to actualize its true potential. A true geek at heart, he spends some of his leisure time writing code in the hope that it may be helpful to the masses. You can find more information on him at <http://mihirmone.appspot.com>.

He is also a motorsport junkie, so you may find him loitering around the race tracks from time to time (especially if Formula 1 is involved).

Takeharu Oshida works at a small start-up, Mobilus (<http://mobilus.co.jp/>). Mobilus provides a real-time communication platform and SDK called Konnect.

As a JavaScript engineer, he designs APIs, writes code and tests, activates EC2 instances, and deploys code. In other words, he is involved in everything, from frontend to backend.

He is also a member of the Xitrum web framework project (<http://xitrum-framework.github.io/>). In this project, he is learning the functional programming style of Scala by creating sample applications or translating documents.

I want to thank to my colleague, Ngoc Dao, who introduced this book to me.

Juri Strumpflohner is a passionate developer who loves to code, follow the latest trends on web development, and share his findings with others. He has been working as a coding architect for an e-government company, where he is responsible for coaching developers, innovating, and making sure that the software meets the desired quality.

Juri strongly believes in the fact that automated testing approaches have a positive impact on software quality and, ultimately, also contribute to the developer's own productivity.

When not coding, Juri is either training or teaching Yoseikan Budo, a martial art form in which he currently owns a 2nd Dan black belt. You can follow him on Twitter at [@juristr](#) or visit his blog at <http://juristr.com> to catch up with him.

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	1
Chapter 1: Welcome to BDD	7
The test-first approach	7
The test-first cycle	9
Write a failing test	10
Make the test pass	10
Clean the code	11
Repeat!	12
Consequences of the test-first cycle	13
BDD versus TDD	14
Exploring unit testing	16
The structure of a test	21
Test doubles	22
What is a good test?	23
Summary	24
Chapter 2: Automating Tests with Mocha, Chai, and Sinon	27
Node and NPM as development platforms	27
Installing Node and NPM	28
Configuring your project with NPM	29
Introducing Mocha	32
Useful options in Mocha	35
Our first test-first cycle	37
More expressive assertions with Chai	41
Working with the "should" interface	45
Red/Green/Refactor	47
Parameterized tests	56
Organizing your setup	58
Defining test scenarios	63

Table of Contents

Test doubles with Sinon	65
Is it traditional TDD or BDD?	70
Welcome Sinon!	71
Integrating Sinon and Chai	73
Summary	76
Chapter 3: Writing BDD Features	77
Introducing myCafé	77
Writing features	78
Displaying a customer's order	79
Tips for writing features	84
Starting to code the scenarios	86
Testing asynchronous features	89
Testing a callback-based API	89
Testing a promise-based API	91
Interlude – promises 101	92
Mocha and promises	95
Organizing our test code	101
The storage object pattern	104
The example factory pattern	108
Finishing the scenario	112
Parameterized scenarios	117
Finishing our feature	120
Summary	127
Chapter 4: Cucumber.js and Gherkin	129
Getting started with Gherkin and Cucumber.js	130
Preparing your project	130
Writing your first scenario in Gherkin	132
Executing Gherkin	134
The World object pattern	138
Better step handlers	143
Better reporting	145
Writing advanced scenarios	147
Gherkin example tables	147
Consolidating steps	153
Advanced setup	156
Gherkin-driven example factory	159
Implicit versus explicit setup	161
The Background section	164
Parameterized scenarios	165
Finishing the feature	170

Table of Contents

Useful Cucumber.js features	172
Tagging features and scenarios	172
Hooks	174
The before hook	174
The after hook	175
The around hook	175
Non-English Gherkin	176
Cucumber.js or Mocha?	176
Summary	177
Chapter 5: Testing a REST Web API	179
The approach	179
A strategy to test web APIs	180
Mocha or Cucumber.js?	182
The plan	182
Testing the GET order feature	183
Exploring our feature a bit	184
Starting, stopping, and setting up our server	186
Testing whether the API responds with 200 Ok	189
Should we use a realistic order object?	190
Implementing the test	191
Testing our HAL resource for orders	193
The contract with the business layer	195
Finishing the scenario	198
Testing slave resources	202
The order actions	202
Testing embedded resources	210
Extracting cross-cutting scenarios	216
Homework!	219
Summary	221
Chapter 6: Testing a UI Using WebDriverJS	223
Our strategy for UI testing	223
Choosing the right tests for the UI	225
The testing architecture	226
WebDriverJS	229
Finding and interacting with elements	231
Complex UI interaction	235
Injecting scripts	236
Command control flows	238
Taking screenshots	240
Working with several tabs and frames	240

Table of Contents

Testing a rich Internet application	241
The setup	242
The test HTML page	242
Serving the HTML page and scripts	244
Using browserify to pack our code	245
Creating a WebDriver session	249
Testing whether our view updates the HTML	249
Testing whether our view reacts with the user	260
What about our UI control logic?	267
Summary	270
Chapter 7: The Page Object Pattern	273
Introducing the Page Object pattern	273
Best practices for page objects	274
A page object for a rich UI	277
Building a page object that reads the DOM	279
Building a page object that interacts with the DOM	284
Testing the navigation	294
Summary	300
Chapter 8: Testing in Several Browsers with Protractor and WebDriver	303
Testing in several browsers with WebDriver	303
Testing with PhantomJS	304
Running in several browsers	307
The Selenium Server	309
Welcome Protractor!	312
Running the tests in parallel	319
Other useful configuration options	320
Using the Protractor API	322
Summary	330
Chapter 9: Testing Against External Systems	333
Writing good test doubles	334
Testing against external systems	336
Testing against a database	337
Accessing the DB directly	337
Treating the DAO as a collection	344
Testing against a third-party system	350
The record-and-replay testing pattern	351
Summary	356

Table of Contents

Chapter 10: Final Thoughts	357
TDD versus BDD	357
A roadmap to BDD	360
BDD versus integration testing	360
BDD is for testing problem domains	361
Concluding the book	364
Next steps	366
Summary	366
Index	367

Preface

JavaScript is not only widely used to create attractive user interfaces for the Web, but, with the advent of Node.js, it is also becoming a very popular and powerful language with which to write server-side applications. In this context, JavaScript systems are no longer toy applications, and their complexity has grown exponentially. To create complex applications that behave correctly, it is almost mandatory to cover these systems with an automated test suite. This is especially true in JavaScript because it does not have a compiler to help developers. Unfortunately, it is easy to fall into testing pitfalls that will make your test suite brittle; hard to maintain, and sooner or later, they will become another headache instead of a solution. Using behavior-driven development and some common testing patterns and best practices, you will be able to avoid these traps.

A lot of people see the whole TDD/BDD approach as a black-and-white decision. Either you do not do it, or you try to achieve a hundred percent test coverage. The real world calls for a more pragmatic approach: write the tests that really pay off and do not write those that do not give you much value. To be able to take this kind of decision, a good knowledge of BDD and the costs associated with it is needed.

What this book covers

Chapter 1, Welcome to BDD, presents the basic concepts that act as a foundation for BDD. Its goal is to debunk a few false preconceptions about BDD and to clarify its nomenclature. It is the only theoretical chapter in the whole book.

Chapter 2, Automating Tests with Mocha, Chai, and Sinon, introduces the basic tools for testing in JavaScript. We will go through the installation process and some simple examples of testing. You can safely skip this chapter if you are well versed with these tools.

Chapter 3, Writing BDD Features, presents some techniques for transforming a functional requirement written in normal language into a set of automated BDD tests or features. We will write our first BDD feature.

Chapter 4, Cucumber.js and Gherkin, repeats the exercise of the previous chapter but this time using Cucumber.js. This way we can compare it fairly with Mocha. You can safely skip this chapter if you already know Cucumber.js.

Chapter 5, Testing a REST Web API, shows you how to test not only the core logic, but also the Node.js server that publishes a Web API. This chapter will be of special interest if you are writing a REST API.

Chapter 6, Testing a UI Using WebDriverJS, shows you how to approach testing the UI layer from the perspective of BDD. You will also learn about WebDriverJS and how it can help you in this task.

Chapter 7, The Page Object Pattern, explains how to create robust UI tests that are less susceptible to being broken by UI design changes. For that, we will apply the page object pattern.

Chapter 8, Testing in Several Browsers with Protractor and WebDriver, shows you how to use the Protractor framework to run your test suite in several browsers.

Chapter 9, Testing Against External Systems, gives you some basic techniques for doing this and, most important, shows you when not to do it. Although this kind of test is not strictly BDD, sometimes you do need to test against external systems.

Chapter 10, Final Thoughts, briefly summarizes the book and clarifies the right granularity for BDD testing. It will also tell you whether to do only BDD at the core business level, or add additional tests at other levels.

What you need for this book

You can follow the code samples in this book using any modern PC or laptop. The code samples should work on Linux and OS X. You can follow the code using Windows, too, but keep in mind that you will need to slightly modify the command-line commands shown in the book to the Windows syntax.

You should have installed at least a modern evergreen web browser, such as Internet Explorer 10 or above (<http://support.microsoft.com/product/internet-explorer/internet-explorer-10/>), Google Chrome (<http://www.google.com/chrome/>), or Firefox (<https://www.mozilla.org/en-US/firefox/new/>).

JavaScript is an interpreted language, so you do not need any special IDE or editor; any editor that supports simple plain text will do. Having said that, I recommend using an advanced editor such as vi, vim, TextMate (<http://macromates.com/>), Sublime (<http://www.sublimetext.com/>), or Atom (<https://atom.io/>). If you prefer an IDE, you can try WebStorm (<https://www.jetbrains.com/webstorm/download/>), although a full-fledged IDE is not needed.

During the book, especially in *Chapter 2, Automating Tests with Mocha, Chai, and Sinon*, detailed explanations are given about how to install and configure the necessary software and tools. This includes Node.js, WebDriver, and all the libraries we are going to use. All of them are open source and free-of-charge.

Who this book is for

This book is for any JavaScript developer who is interested in producing well-tested code. If you have no prior experience with testing Node.js, or any other tool, do not worry as they will be explained from scratch. Even if you have already used some of the tools explored in the book it can still help you to learn additional testing techniques and best practices.

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:
"The headers property of the `replay` object is an array of regular expressions."

A block of code is set as follows:

```
var result = b.operation(1, 2);

expect(result).to.be.deep.equal({
  args: [1, 2],
  result: 3
});
```

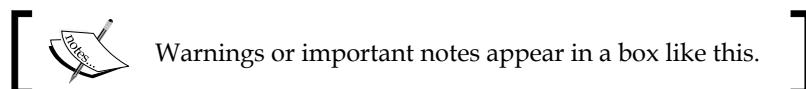
When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
module.exports = function (findConfiguration) {
  return function () {
    findConfiguration('default');
    return validatorWith([
      nonPositiveValidationRule,
      nonDivisibleValidationRule(3, 'error.three'),
      nonDivisibleValidationRule(5, 'error.five')
    ]);
  };
};
```

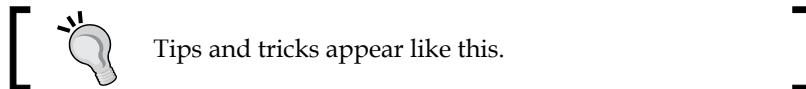
Any command-line input or output is written as follows:

```
$ me@~-> mkdir validator
$ me@~-> cd validator
$ me@~/validator> npm init
```

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: "You can download and execute a nice installer by going to Node.js website and clicking on **Install**."



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

Welcome to BDD

Before we start coding tests, we need to understand what **behavior-driven development (BDD)** is and how it differs from **test-driven development (TDD)**.

We need to understand not only the concept of BDD, but also all the jargon associated with it. For example, what is a feature? Or what is a unit test? So, in this chapter, I will try to clarify some common vocabulary in order to give you a solid understanding of what every technical term means.

In this chapter, you will learn:

- The reason for writing automated tests
- The workflow prescribed by the test-first approach
- What BDD is and how it differs from TDD
- What a unit test really is
- The different phases that compose a test
- What test doubles are and the different kinds of test doubles that exist
- The characteristics of a good test

The test-first approach

Testing is nothing new in software engineering; in fact, it is a practice that has been implemented right from the inception of the software industry, and I am not talking only about manual testing, but about automated testing as well. The practice of having a set of automated tests is not exclusive to TDD and BDD, but it is quite old. What really sets apart approaches such as TDD and BDD is the fact that they are test-first approaches.

In traditional testing, you write your automated test after the code has been written. At first sight, this practice seems to be common sense. After all, the point of testing is discovering bugs in the code you write, right? Probably, these tests are executed by a different team than the one that wrote the code in order to prevent the development team from cheating.

Behind this traditional approach lies the following assumptions:

- Automated tests can discover new bugs
- The project is managed under a waterfall life cycle or similar, where large chunks of functionality are developed until they are *perfect*, and only then is the code deployed

These assumptions are mostly false nowadays. Automated tests cannot discover anything new but only provide feedback about whether the code behaves as specified or expected. There can be errors in the specification, misunderstandings, or simply different expectations of what is correct between different people. From the point of view of preventing bugs, automated tests are only good as **regression test suites**. A regression test suite contains tests that prove that a bug that is already known is fixed. Since there usually exists a lot of misunderstanding between the stakeholders themselves and the development team, the actual discovery of most bugs is often done during exploratory testing or by actual users during the beta or alpha phase of the product.

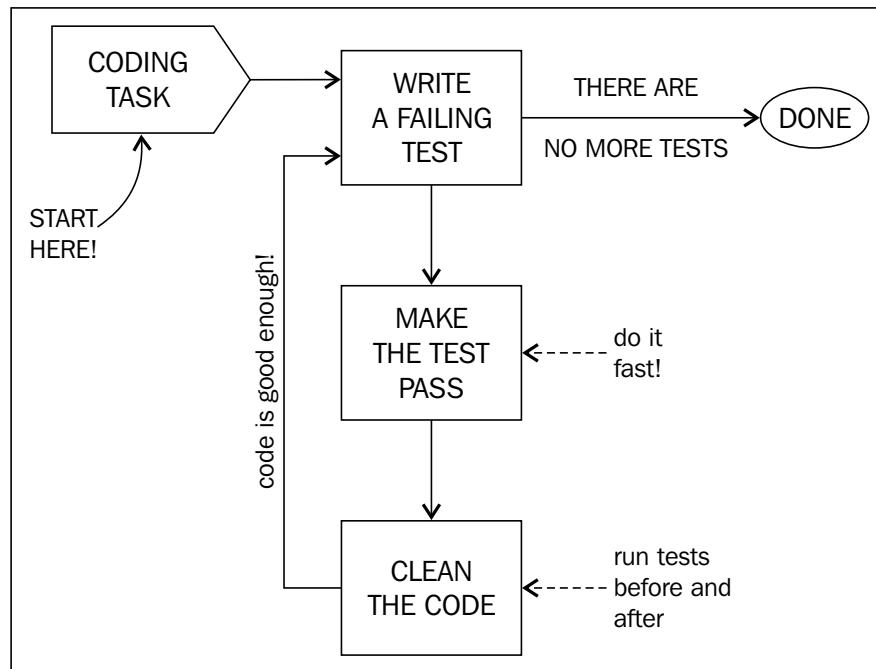
About the waterfall approach, the industry has been moving away from it for some time now. It is clearly understood that not only fast time to market is crucial, but that a project's target can undergo several changes during the development phase. So, the requirements cannot be specified and set in stone at the beginning of the project. To solve these problems, the agile methodologies appeared, and now, they are starting to be widely applied.

Agile methodologies are all about fast feedback loops: plan a small slice of the product, implement it, and deploy and check whether everything is as expected. If everything is correct, at least we would already have some functionality in production, so we could start getting some form of benefit from it and learn how the user engages with the product. If there is an error or misunderstanding, we could learn from it and do it better in the next cycle. The smaller the slice of the product we implement, the faster we will iterate throughout the cycles and the faster we will learn and adapt to changes. So ideally, it is better to build the product in small increments to be able to obtain the best from these feedback loops.

This way of building software changed the game, and now, the development team needs to be able to deliver software with a fast pace and in an incremental way. So, any good engineering practice should be able to enable the team to change an existing code base quickly, no matter how big it is, without a detailed full plan of the project.

The test-first cycle

In this context, the test-first approach performs much better than the traditional one. To understand why, first, let's have a look at the test-first cycle:



As you can see, the cycle starts with a new coding task that represents any sensible reason to change the codebase. For example, a new functionality or a change in an existing one can generate a new coding task, but it can also be triggered by a bug. We will talk a bit more in the next section about when a new coding task should trigger a new test-first cycle.

Write a failing test

Once we have a coding task, we can engage in a test-first cycle. In the first box of the previous diagram, **write a failing test**, we try to figure out which one is the simplest test that can fail; then, we write it and finally see it fail.

Do not try to write a complex test; just have patience and go in small incremental steps. After all, the goal is to write the simplest test. For this, it is often useful to think of the simplest input to your system that will not behave as expected. You will often be surprised about how a small set of simple tests can define your system!

Although we will see this in more detail in the upcoming chapters, let me introduce a small example. Suppose we are writing the validation logic of a form input that takes an e-mail and returns an array of error messages. According to the test-first cycle, we should start writing the most simple test that could fail, and we still have not written any production code. My first test will be the success case; we will pass a valid e-mail and expect the validation function to return an empty array. This is simple because it establishes an example of what is valid input, and the input and expectations are simple enough.

Make the test pass

Once you have a failing test, you are allowed to write some production code to fix it. The point of all of this is that you should not write new code if there is not a good reason to do so. In test-first, we use failing tests as a guide to know whether there is need for new code or not. The rule is easy: *you should only write code to fix a failing test or to write a new failing test.*

So, the next activity in the diagram, **make the test pass**, means simply to write the required code to make the test pass. The idea here is that you just write the code as fast as you can, making minimal changes needed to make the test pass. You should not try to write a nice algorithm or very clean code to solve the whole problem. This will come later. You should only try to fix the test, even if the code you end up writing seems a bit silly. When you are done, run all the tests again. Maybe the test is not yet fixed as you expected, or your changes have broken another test.

In the example of e-mail validation, a simple return statement with a empty array literal will make the test pass.

Clean the code

When all the tests are passing, you can perform the next activity, **clean the code**. In this activity, you just stop and think whether your code is good enough or whether it needs to be cleaned or redesigned. Whenever you change the code, you need to run all the tests again to check that they are all passing and you have not broken anything. Do not forget that you need to clean your test code too; after all, you are going to make a lot of changes in your test code, so it should be clean.

How do we know whether our code needs some cleaning? Most developers use their intuition, but I recommend that you use good, established design principles to decide whether your code is good enough or not. There are a lot of established design principles around, such as the SOLID principles (see http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf) or Craig Larman's GRASP patterns (see http://www.craiglarman.com/wiki/index.php?title=Books_by_Craig_Larman#Applying_UML_and_Patterns). Unfortunately, none of the code samples of these books are in JavaScript, so I will summarize the main ideas behind these principles here:

- Your code must be readable. This means that your teammates or any software engineer who will read your code 3 months later should be able to understand the intent of the code and how it works. This involves techniques such as good naming, avoiding deep-nested control structures, and so on.
- Avoid duplication. If you have duplicated code, you should refactor it to a common method, class, or package. This will avoid double maintenance whenever you need to change or fix the code.
- Each code artifact should have a single responsibility. Do not write a function or a class that tries to do too much. Keep your functions and objects small and focused on a single task.
- Minimize dependencies between software components. The less a component needs to know about others, the better. To do so, you can encapsulate internal state and implementation details and favor the designs that interchange less information between components.
- Do not mix levels of abstractions in the same component; be consistent in the language and the kind of responsibility each component has.

To clean your code, you should apply small refactoring steps. Refactoring consists of a code change that does not alter the functionality of the system, so the tests should always pass before and after each refactoring session. The topic of refactoring is very big and out of the scope of this book, but if you want to know more about it, I recommend *Refactoring: Improving the Design of Existing Code* (<http://martinfowler.com/books/refactoring.html>).

Anyway, developers often have a good instinct to make their code *better*, and this is normally just enough to perform the clean code step of the test-first cycle. Just remember to do this in small steps, and make sure that your tests pass before and after the refactoring.



In a real project, there will be times when you just do not have much time to clean your code, or simply, you know there is something wrong with it, but you cannot figure out how to clean it at that moment. In such occasions, just add a *TODO* comment in your code to mark it as technical debt, and leave it. You can talk about how to solve the technical debt later with the whole team, or perhaps, some iterations later, you will discover how to make it better.

Repeat!

When the code is good enough for you, then the cycle will end. It is time to start from the beginning again and write a new failing test. To make progress, we need to prove with a failing test whether our own code is broken!

In our example, the code is very simple, so we do not need to clean up anything. We can go back to writing a failing test. What is the most simple test that can make our code fail? In this case, I would say that the empty string is an invalid e-mail, and we expect to receive an **email cannot be empty** error. This is a very simple test because we are only checking for one kind of error, and the input is very simple; an empty string.

After passing this test, we can try to introduce more tests for other kinds of errors. I would suggest the following order, by complexity:

- Check for the presence of an @ symbol
- Check for the presence of a username (@mailcompany.com should fail, for example)
- Check for the presence of a domain (peter@ should fail too)
- Check whether the domain is correct (peter@bad#domain!.com should fail)

After all of these tests, we would probably end up with a bunch of `if` statements in our code. It is time to refactor to remove them. We can use a regular expression or, even better, have an array or validation rules that we can run against the input.

Finally, after we have all the rules in place and our code looks clean, we can add a test to check for several errors at the same time, for example, checking that `@bad#domain!.com` should return an array with the **missing username** and **incorrect domain** errors.

What if we cannot write a new failing test? Then, we are simply done with the coding task!

As a summary, the following are the five rules of the test-first approach:

- Don't write any new tests if there is not a new coding task.
- A new test must always fail.
- A new test should be as simple as possible.
- Write only the minimum necessary code to fix a failing test, and don't bother with quality during this activity.
- You can only clean or redesign your code if all the tests pass. Try to do it in each cycle if possible.

Consequences of the test-first cycle

This way of writing code looks weird at first and requires a lot of discipline from the engineers. Some people think that it really adds a big overhead to the costs of a project. Maybe this is true for small projects or prototypes, but in general, it is not true, especially for codebases that need to be maintained during periods of over 3 or 4 months.

Before test-first, most developers were doing manual testing anyway after each change they made to the code. This manual testing was normally very expensive to achieve, so test-first is just cutting costs by automating such activity and putting a lot of discipline in our workflow.

Apart from this, the following are some subtle consequences:

- Since you write tests first, the resulting code design ends up being easily testable. This is important since you want to add tests for new bugs and make sure that changes do not break the old functionality (regression).

- The resulting codebase is minimal. The whole cycle is designed to make us write just the amount of code needed to implement the required functionality. The required functionality is represented by failing tests, and you cannot write new code without a failing test. This is good, because the smaller the code base is, the cheaper it is to maintain.
- The codebase can be enhanced using refactoring mechanisms. Without tests, it is very difficult to do this, since you cannot know whether the code change you have done has changed the functionality.
- Cleaning the code in each cycle makes the codebase more maintainable. It is much cheaper to change the code frequently and in small increments than to do it seldom and in a big-bang fashion. It is like tidying up your house; it is better to do it frequently than do it only when you expect guests.
- There is fast feedback for the developers. By just running the test suite, you know, in the moment, that the changes in the code are not breaking anything and that you are evolving the system in a good direction.
- Since there are tests covering the code, the developers feel more comfortable adding features to the code, fixing bugs, or exploring new designs.

There is, perhaps, a drawback: you cannot adopt the test-first approach easily in a project that is in the middle of its development and has been started without this approach in mind. Code written without a test-first approach is often very hard to test!

BDD versus TDD

The test-first approach covered in the previous section is what has been described generically as TDD.

The problem with TDD, as already presented, is that it does not say anything about what a coding task is, when a new one should be created, or what kind of changes we should allow.

It is clear that a change in a requisite or a newly discovered bug should trigger a TDD cycle and involve a new coding task. However, some practitioners think that it is also OK to change the codebase, because some engineer thought that a change in the technical design would be good for the system.

The biggest problem in classic TDD is that there is a disconnection between what the product is supposed to do and what the test suite that the development team builds is testing. TDD does not explicitly say how to connect both worlds. This leads to a lot of teams doing TDD, but testing the wrong things. Yes, perhaps they were able to test all their classes, but they tested whether the classes behave as expected, not whether the product behaves as expected.

Yes, perhaps they have a very detailed test suite with high coverage and with all its tests passing, but this offers no clue about whether the product itself will work as expected or whether a bug is resolved. This is a bad situation, as the main benefit of the tests is in the fast feedback they provide.

BDD tries to fix these problems by making the test suite directly dependent of the feature set of the product. Basically, BDD is a test-first approach where a new coding task can be created only when a change in the product happens: a new requisite, a change in an existing one, or a new bug.

This clarification changes rule 1 of test-first, from *Don't write any new tests if there is not a new coding task* to *Don't write any new tests if there is not a change in the product*. This has some important implications, as follows:

- You should not add a new class or function or change the design if there is not a change in the product. This is a more specific assertion about coding tasks than the generic one about TDD.
- As a change in the product always represents only a feature or bug, you only need to test features or bugs, not components or classes. There is no need to test individual classes or functions. Although this does not mean that it is a bad idea to do so, such tests are not viewed as essential from the BDD point of view.
- Tests are always about describing how the product behaves and never about technical details. This is a key difference with TDD.
- Tests should be described in a way that the stakeholders can understand to give feedback about whether they reflect their expected behavior of the system. That is why, in BDD jargon, tests are not called tests, but specifications or features.
- Test reports should be understandable for the stakeholders. This way, they can have direct feedback of the status of the project, instead of having the need for the chief architect to explain the test suite result to them.
- BDD is not only an engineering practice, but it needs the team to engage frequently with the stakeholders to build a common understanding of the features. If not, there would be a big risk that we are testing the wrong feature.

Of course, there were teams that practiced TDD in this way, avoiding all of the problems mentioned earlier. However, it was Dan North who first coined the term BDD to this specific way of doing TDD and to popularize this way of working.

BDD exposes a good insight: we should test features instead of components. This is very important from the perspective of how to design a good test suite. Let's explore this subject a bit in the next section.

Exploring unit testing

99.99 percent of the projects we are going to face will be complex and cannot be tested with a single test. Even small functionalities that a non-engineer would consider very simple will actually be more complex than expected and have several corner cases. This forces us to think about how to decompose our system in tests or, in other words, what exactly are the tests that we should write.

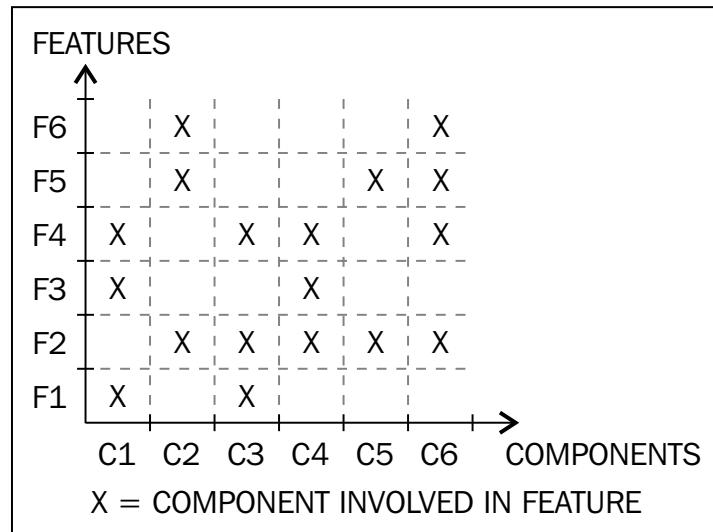
In the beginning of the test-first movement, there was no clear answer to this question. The only guidance was to write a test for each unit and make the tests from different units independent between them.

The notion of units is very generic and does not seem to be very useful to guide the practice of test-first. After a long debate in the community, it seems that there is a consensus that there exists at least two kinds of units: features and components.

A feature is a single concrete action that the user can perform on the system; this will change the state of the system and/or make the system perform actions on other third-party systems. Note that a feature is usually a small-grained piece of functionality of the system, and a use case or user story can map to several features. An important thing about features is that they describe the behavior of the system from the point of view of the user. Slicing a user story into features is a key activity of BDD, and throughout the book, we will see plenty of examples of how to do it.

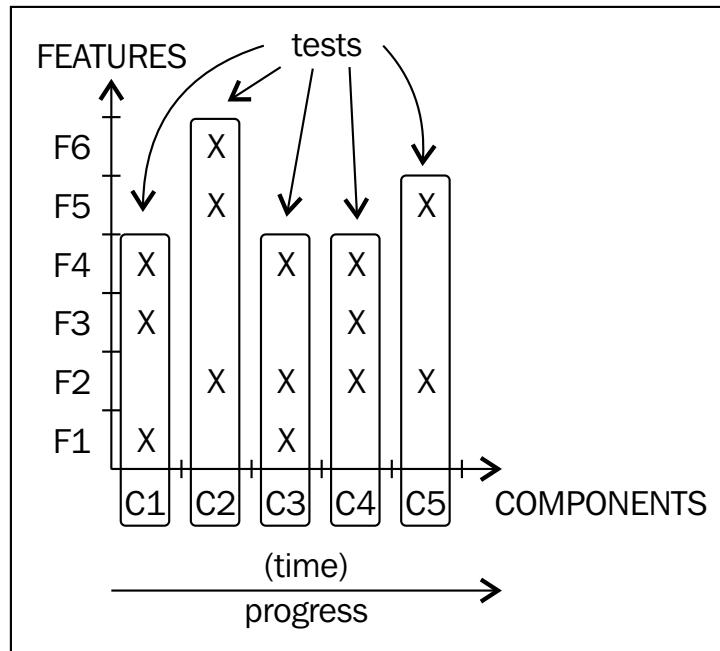
The other kinds of units are the components of our system. A component is any software artifact, such as classes, procedures, or first-order functions, that we use to build the system.

We can conceptualize any product we are building as a matrix of features versus components, like in the following image:



In this image, we can see that any system implements a set of features, and it is implemented by a set of components. The interesting thing is that there is seldom a one-to-one relationship between components and features. A single feature involves several components, and a single component can be reused across several features.

With all this in mind, we can try to understand what traditional TDD, or traditional unit testing, is doing. In the traditional approach, the idea is to make unit tests of components. So, each component should have a test of its own. Let's have a look at how it works:



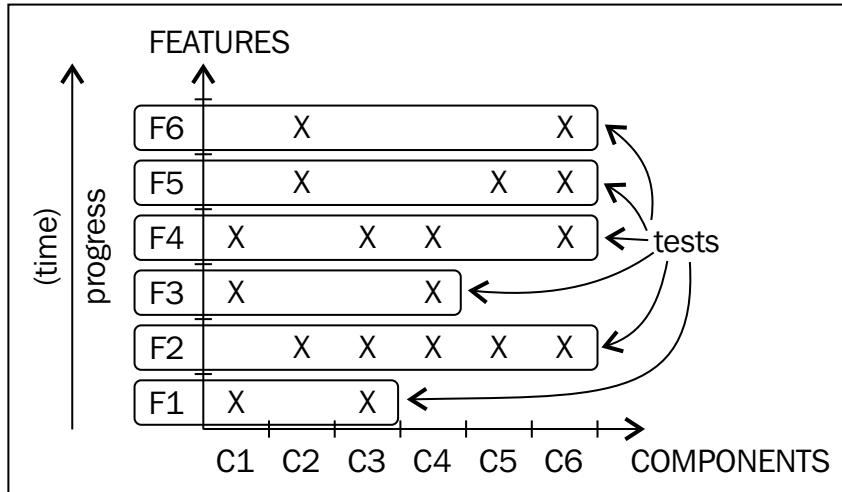
In the preceding image, you can see that the system is built incrementally, one component at a time. The idea is that with each increment, a new component is created or an existing one is upgraded in order to support the features. This has the advantage that if a test is failing, we know exactly which component is failing.

Although this approach works in theory, in practice, it has some problems. Since we are not using the features to guide our tests, they can only express the expected behavior of the components. This usually generates some important problems, such as the following ones:

- There is no clear and explicit correlation between the components and the features; in fact, this relationship can change over time whenever there is a design change. So, there is no clear progress feedback from the test suite.
- The test results only make sense for the engineering team, since it is all about components and not the behavior of the systems. If a component test is failing, which features are failing? Since there is not a clear correlation between features and components, it is expensive to answer this question.
- If there is a bug, we don't know which tests to modify. Probably, we will need to change several tests to expose a single bug.
- Usually, you will need to put a lot more effort into your technical design to have a plan of what components need to be built next and how they fit together.
- The tests are checking whether the component behaves according to the technical design, so if you change the design, then you need to change the tests. The whole test suite is vulnerable to changes in the design, making changes in the design harder. Hence, a needed refactor is usually skipped, making the whole quality of the codebase worse and worse as time passes.

Of course, a good and experienced engineering team can be successful with this approach, but it is difficult. It is not surprising that a lot of people are being very vocal against the test-first approach. Unit test components is the classic and de facto approach to test-first, so when someone says terms such as TDD or unit testing, they usually mean component unit testing. This is why problems with component unit testing have been wrongly confused with problems of the general test-first approach.

The other way of doing test-first is to unit test features, which is exactly what BDD make us do. We can have a look at the diagram to see how a system progresses using BDD:



As we can see, as time progresses, we add tests for each feature, so we can have good feedback about the status of completion of the project. If a test is failing, then it means that the corresponding feature is broken.

On the other hand, we don't need a very detailed up-front design to start coding. After all, we have the guidance of the behavior of the system to start the test-first workflow, and we can fine-tune our design incrementally using the "clean code" step of the workflow. We can discover components on the fly while we are delivering features to the customer. Only a high-level architecture, some common conventions, and the set of tools and libraries to use, are needed before starting the development phase. Furthermore, we can isolate the test from most technical changes and refactorings, so in the end, it will be better for our codebase quality.

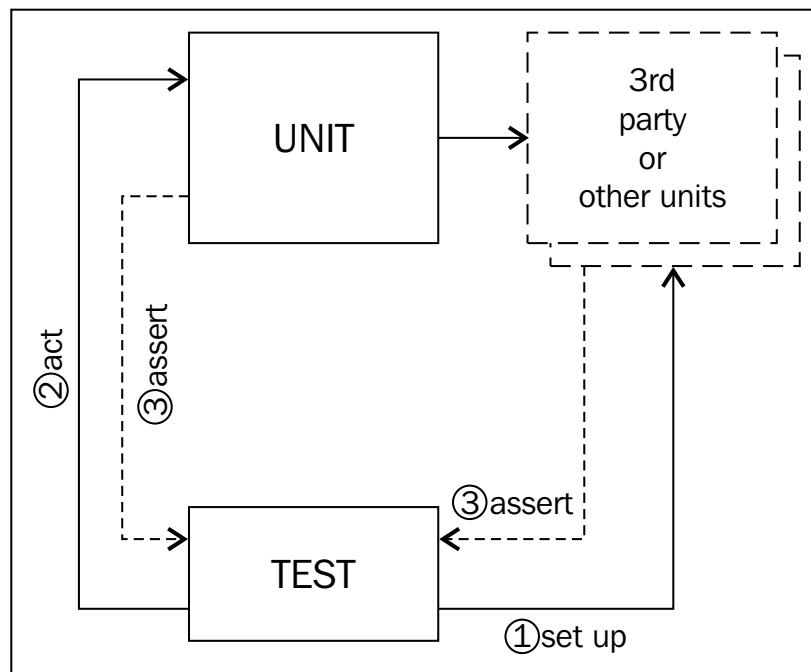
Finally, it seems to be common sense to focus on the features; after all, this is what the customer is really paying us for. Features are the main thing we need to ensure that are working properly. An increment in component unit testing does not need to deliver any value, since it is only a new class, but an increment in BDD delivers value, since it is a feature. It does not matter whether it is a small feature or not; it is a tangible step forward in project termination.

There is, of course, a disadvantage in this approach. If a test is failing, we know which feature is failing, but we do not know which component needs to be fixed. This involves some debugging. This is not a problem for small and medium systems, since a feature is usually implemented by 3–5 components. However, in big systems, locating the affected component can be very costly.

There is no silver bullet. In my opinion, BDD is an absolute minimum, but unit testing some of the key components can be beneficial. The bigger the system is, the more component unit testing we should write, in addition to the BDD test suite.

The structure of a test

As we saw earlier, a unit could be a feature if we are doing BDD, or it could be a component if we are doing traditional TDD. So, what does a unit test look like? From a very high level point of view, a unit test is like the following image:



You can see that the test is acting as the unit. The term "act" means that the test is performing a single operation on the unit through its public API.

Then, the test must check or assert the result of the operation. In this phase, we need to check whether the actual return value is as we expect, but we also need to check whether the side effects are the expected ones. A side effect is a message that the unit sends to other units or third-party systems in order to perform the action correctly.

Side effects look quite abstract, but in fact, they are very simple. For example, from the point of view of traditional TDD, a side effect can be a simple call from one class to another. From the point of view of BDD, a side effect can be a call to a third-party system, such as an SMS service, or a write to the database.

The result of an action will depend on the prior state of the system we are testing. It is normal that the expected result of the very same action varies according to the specific state the system is in. So, in order to write a test, we need to first set up or arrange the system in a well-known state. This way, our test will be repeatable.

To sum up, every test must have the following three phases:

- **Set up/Arrange:** In this phase, we set up the state of the system in a well-known state. This implies choosing the correct input parameters, setting up the correct data in the database, or making the third-party systems return a well-known response.
- **Act:** In this phase, we perform the operation we are testing. As a general rule, the act phase of each test should involve only one action.
- **Assert:** In this phase, we check the return value of the operation and the side effects.

Test doubles

Whenever we see the term "unit testing", it means that we are making tests of the units of our system in an isolated way. By isolated, I mean that each test must check each unit in a way independent of the others. The idea is that if there is a problem with a unit, only the tests for that unit should be failing, not the other ones. In BDD, this means that a problem in a feature should only make the tests fail for that feature. In component unit testing, it means that a problem with a component (a class, for example) should only affect the tests for that class. That is why we prescribe that the act phase should involve only one action; this way, we do not mix behaviors.

However, in practice, this is not enough. Usually, features can be chained together to perform a user workflow, and components can depend on other components to implement a feature.

This is not the only problem, as we saw earlier; it is usually the case that a feature needs to talk with other systems. This implies that the set up phase must manipulate the state of these third-party systems. It is often unfeasible to do so, because these systems are not under our control. Furthermore, it can happen that these systems are not really stable or are shared by other systems apart from us.

In order to solve both the isolation problem and the set up problem, we can use test doubles. Test doubles are objects that impersonate the real third-party systems or components, just for the purpose of testing. There are mainly the following type of test doubles:

- **Fakes:** These are a simplified implementation of the system we are impersonating. They usually involve writing some simple logic. This logic should never be complex; if not, we will end up reimplementing such third-party systems.
- **Stubs:** These are objects that return a predefined value whenever one of its methods is called with a specific set of parameters. You can think of them as a set of hardcoded responses.
- **Spies:** These are objects that record their interactions with our unit. This way, we can ask them later what happened during our assertion phase.
- **Mocks:** These are self-validating spies that can be programmed during the set up phase with the expected interaction. If some interaction happens that is not expected, they would fail during the assertion phase.

We can use spies in the assertion phase of the test and stubs in the set up phase, so it is common that a test double is both a spy and a stub.

In this book, we will mostly use the first three types, but not mocks, so don't worry much about them. We will see plenty of examples for them in the rest of the book.

What is a good test?

When we are writing tests, we need to keep in mind a series of guidelines in order to end up with a useful test suite. Every test we write should have the following properties:

- They should be relevant. A test must be relevant from the point of view of the product. There is no point in testing something that, when it is done, does not clearly move the project forward to completion. This is automatically achieved by BDD, but not by traditional TDD.

- They should be repeatable. Tests must always offer the same results if there has not been a code change. If it is failing, you must change the code to see it pass, and if it is passing, it must not start failing if nobody changed the code. This is achieved through a correct setup of the system and the use of test doubles. If tests are not repeatable, they offer no useful information! I have seen teams ignore tests that are flipping between passing and failing because of incorrect setup or race conditions. It would have been better not to waste time and money in writing a test that nobody cares about because it is not reliable.
- They should be fast. After all, one key point of test-first is rapid feedback and quick iteration. It is not very cost effective if you need to wait 15 minutes for the tests to end whenever you make a code change in a test-first cycle.
- They should be isolated. A test should fail only because the feature (or component) it is testing has a defect. This will help us diagnose the system to pinpoint where the error is. This will help us write code in an incremental fashion in the order our customers require (often, the most valuable features first). If the test is not isolated, then we often cannot write a new test, because we need first to write another feature or component that this one depends on.

Summary

The test-first approach appeared as an engineering practice to back up the agile methodologies. It supports the notion of incremental design and implementation of the codebase in order to be able to deliver software fast, incrementally, and in short iterations.

The test-first approach tells us to first write the most simple failing test that we can think of, fix it with the smallest change of code possible, and finally, clean the code, changing the design if necessary and taking advantage of the fact that we have tests as a safety net. Repeat the cycle until there is no new failing test to write.

There are two main approaches to test-first: traditional TDD and BDD. In traditional TDD, or component unit testing, we test components (classes, functions, and so on) in isolation from other components. In BDD, we test simple user actions on the system, also known as features, in isolation from other features. Both are forms of unit testing, but due to historic reasons, we reserve the term "unit testing" for component unit testing.

In my opinion, the BDD approach is superior, because it relates the tests with the actual behavior of the system, making the progress of the project more visible, focusing the team on what really matters and decoupling the tests themselves from the specific details of the technical design. However, in big systems, it can be difficult to diagnose which components should be fixed when a feature fails, so some degree of traditional TDD is still useful.

Tests should be isolated to avoid coupling between them and enable fast detection of which feature/component must be fixed. They should also be fast to get a quick feedback cycle during development. Furthermore, tests should be repeatable; if not, we cannot trust their result, and they become a waste of time and money.

To make tests isolated, fast, and repeatable, we can use test doubles. They replace and impersonate third-party systems or components in our test suite. They can be used both to set up the system in a predictable way, hence achieving repeatability and quick execution, and to check the side effects produced by the system under test. In traditional unit testing, we can use them to isolate the component under test from other components.

This concludes the first chapter. Fortunately, it is the only one devoted to theory in this book. In the next chapter we will start coding!

2

Automating Tests with Mocha, Chai, and Sinon

Before we start making some BDD, let's familiarize ourselves with the basic tools available in JavaScript to write and execute a test. In this chapter, we will explore the main capabilities of Mocha, the most popular test runner in JavaScript. We will perform the following tasks:

- Writing expressive assertions using the Chai package
- Creating test doubles using the Sinon and sinon-chai packages
- Exploring the basic techniques for organizing our test codebase

To achieve these goals, we will perform a small code kata, or coding exercise, where we will be able to practice not only the tools, but also the test-first cycle explained in the previous chapter.

Node and NPM as development platforms

All the tools that we will use are written in JavaScript. A long time ago, the only way to execute JavaScript was to use a browser, but those days are long gone. Nowadays, we can execute our development tools from the command line during our normal development cycle or from a **Continuous Integration (CI)** server whenever we commit our changes to a source code repository.

The most easy and productive way to run our test tools is to use **Node**. Node is a lightweight and highly-scalable platform for JavaScript, written on top of the excellent V8 JavaScript virtual machine. Node is especially good for applications that perform high-volume IO, but it can also be used as a development platform, as we will see in a moment.

Installing Node and NPM

The examples in this book will work with Node Version 0.10.x or above. If you don't have it installed already, you must do so to follow the code examples.



It is recommended that you do not install a version of Node whose minor version number is not even. These versions are not stable and are experimental. For example, 0.11.x, 0.9.x, and so on, are all unstable versions. As a rule of thumb, you should install the latest even version number.

The following are the three ways to install Node:

- You can download and execute a nice installer by going to <http://nodejs.org/> and clicking on **Install**. This will detect your OS and decide automatically which installer to download. If you want to decide yourself, go to <http://nodejs.org/download/>.
- If you prefer to install Node using a package manager, especially if you are using LINUX, go to <https://github.com/joyent/node/wiki/Installing-Node.js-via-package-manager> and follow the instructions for your OS.
- Another option is to install **Node Version Manager (NVM)**. NVM is a utility that allows you to have several different versions of Node on your machine, switch from one version to another easily, and install new versions. To install NVM, just go to <https://github.com/creationix/nvm> and follow the instructions. Once you have NVM successfully installed, you just need to issue the following command line:

```
$ me@~> nvm install 0.10  
$ me@~> nvm alias default 0.10  
$ me@~> nvm use default
```



I actually use NVM because I have several projects in my development machine that were designed to run on production with different versions of Node. So, I just need to switch to the correct Node version with an `nvm use` command to switch from one project to another.



NVM is not the only version manager for Node. You can try others such as Nodebrew (<https://github.com/hokaccha/nodebrew>).

Configuring your project with NPM

With the installation of Node, the **Node Package Manager (NPM)** is also installed. NPM is actually the utility that we will use for our normal development cycle.

NPM allows you to install libraries, manage the dependencies of your projects, and define a set of commands to build your code.

To start, just create a new folder for your project and initialize it from there:

```
$ me@~> mkdir validator  
$ me@~> cd validator  
$ me@~/validator> npm init
```

The last command, `npm init`, will invoke NPM to generate `package.json` inside the current directory; during the process, it will ask you some questions. Most of them are self-explanatory, but if you do not know what to answer just press `ENTER`. Do not worry; you will be able to edit the `package.json` file later. This is my `package.json` file:

```
{  
  "name": "validator",  
  "version": "0.1.0",  
  "description": "A validation service for Weird LTD.",  
  "main": "index.js",  
  "scripts": {  
    "test": "echo \\\"Error: no test specified\\\" && exit 1"  
  },  
  "author": "Enrique Amodeo",  
  "license": "MIT"  
}
```

This file contains all the information needed for NPM to manage your new package; in fact, this information is what marks the folder as a package. It contains the name of your package, which other packages your package depends on, and whether these dependencies are necessary only in runtime or during development. It has information on the author of the package, the version, the repository where the original code is hosted, the license, and so on. This will be useful if you want to publish your package so that other people can reuse it in their projects.

[ For an exhaustive explanation of all the information that package.json can contain, just issue the `npm help json` command or go to <https://www.npmjs.org/doc/files/package.json.html>. For an exhaustive list of all the commands of NPM, go to <https://www.npmjs.org/doc/cli/npm.html>. In this book, we will only see the options and commands that you need to do BDD.]

Instead of using `npm init`, you could have simply created and edited `package.json`. In fact, you can always edit this file instead of using NPM to do so.

[ However, note that `package.json` must always contain a single valid JSON document. JSON documents do not contain comments or code, and all the strings are defined using double quotes.]

To finish the initialization of the project, you just need to install the tools that we will use: Mocha and Chai. To do so, issue the following command:

```
$ me@~/validator> npm install mocha chai --save-dev
```

This command will install Mocha and Chai locally to your project. This means that it will download the packages and all their dependencies, compile them if necessary, and store the result in a `node_modules/` directory inside the project. In fact, you should now have `node_modules` in your project that contains a folder for Mocha and another for Chai.

[ The `npm install package_name` command will always install the latest stable version. To install a specific version, use `npm install package_name@version` syntax. For example, if we wish to install version 1.10.0 of chai, then we could use the following command:

```
$ me@~/validator> npm install chai@1.10.0 --save-dev
```

Besides all of this, your `package.json` file should now look like this:

```
{  
  "name": "validator",  
  "version": "0.1.0",  
  "description": "A validation service for Weird LTD.",  
  "main": "index.js",  
  "scripts": {  
    "test": "echo \\\"Error: no test specified\\\" && exit 1"  
  },  
  "author": "Enrique Amodeo",
```

```

    "license": "MIT",
    "devDependencies": {
      "chai": "^1.9.1",
      "mocha": "^1.20.1"
    }
  }
}

```

Our package.json file has changed; now it contains a **"devDependencies"** section! This is because we specified the `--save-dev` parameter in the preceding command. This means that, after downloading and installing the packages, NPM will modify the package.json file to specify that your package depends on the Mocha and Chai packages during development time. To understand exactly what this means, let's do a short experiment. First, remove the entire node_modules folder:

```
$ me@~/validator> rm -rf ./node_modules
```

Now, issue the following command:

```
$ me@~/validator> npm install
```

The result is that the packages are reinstalled again, and the node_modules folder is recreated. NPM will just read the information contained in package.json to know which modules to install.

Note that, in this case, NPM will install the latest version that is compatible with the ones specified inside package.json. In the preceding example, we have "`^1.9.1`" and "`^1.20.1`". The `^` character at the beginning specifies that NPM will install the latest version of the package that does not change the leftmost nonzero digit. In this specific case, we will allow patch and minor version changes. We can edit the package.json file to define any kind of version range; see <https://www.npmjs.org/doc/misc/semver.html> for more details.



You should ignore the node_modules folder and not include it into the source control. Whenever you check out the source code, you just need to execute `npm install`, and all the latest and compatible versions of your dependencies will be installed into your project. Including node_modules in your source control is a bad idea, as it is redundant information, makes your repository bigger, and can result in certain problems. After all, each package in node_modules can contain not only JavaScript, but also binary files. Some Node packages contain C code modules that are compiled during installation. The resulting binary files, as opposed to the JavaScript ones, are not cross-platform. You can have a look at <https://github.com/github/gitignore/blob/master/Node.gitignore> to see an example of a typical .gitignore file.

Note that you could have installed the modules using the `--save` option instead of `--save-dev`. This would have installed the packages as runtime dependencies instead of development dependencies. Runtimes dependencies are specified in the "dependencies" section of the package.json file. Although installing Mocha and Chai as runtime dependencies will not generate any errors while replicating the examples in this book, it is not a really good idea. When the `npm install` command installs a package, it will also install all the runtime dependencies of that package, but not the development dependencies. So, if we install Mocha and Chai as runtime dependencies, the other packages that depend on our package will also install Mocha and Chai, even if they are not really needed for the correct execution of our package.

Introducing Mocha

Mocha is a modern test runner that can be executed from Node as well as inside a browser. As we saw earlier, our main approach is to use Node. In this book, I am using Mocha Version 1.20.1, but any 1.x version should be OK.

We have already installed Mocha using NPM, so we can run it with the following command:

```
$ me@~/validator> ./node_modules/.bin/mocha -u bdd -R spec -t 500  
--recursive
```

It fails because we do not have any test yet; we will fix it later. I will explain the exact meaning of these parameters in a moment. For now, note that, to execute Mocha, we need to find out where NPM has installed the executable of the tool. A package can be a simple library, or it can also contain an executable command-line tool, as is the case with Mocha. NPM always will install this executable in the `node_modules/.bin/` folder; hence, we need to execute `./node_modules/.bin/mocha` in order to invoke Mocha.

This would not have been necessary if we had installed Mocha as a global package using the `-g` option:

```
$ me@~/validator> npm install mocha -g
```

This would have installed Mocha as a utility tool available in our global PATH. So, we would have only needed to use the following command:

```
$ me@~/validator> mocha -u bdd -R spec -t 500 --recursive
```

Much better, right? However, actually this approach is not as nice as it seems. The fact is that global packages have a couple of problems:

- They are not included in the dependency management of our package. The package `.json` file will not be modified.
- It's possible that you have several different projects, each one using a different and incompatible version of Mocha. In this case, you will have a big problem if you are using Mocha as a global package.

Due to these problems, it is usually preferable to install any tool we need as a local development dependency. This way, each project can use different versions for each package and tools, without interfering between them, and we can manage all of our dependencies using only the `package.json` file. The only reason to install a global package is when we really want to install a Node-based application, such as a text editor or network monitor, and not for development tools.

So, are we doomed to always use the ugly `./node_modules/.bin/mocha` file to execute Mocha? No, fortunately, there is a solution. You just need to modify the `package.json` file to specify a test command such as the following one:

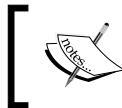
```
{  
  "name": "validator",  
  "version": "0.1.0",  
  "description": "A validation service for Weird LTD.",  
  "main": "index.js",  
  "scripts": {  
    "test": "mocha -u bdd -R spec -t 500 --recursive"  
  },  
  "author": "Enrique Amodeo",  
  "license": "MIT",  
  "devDependencies": {  
    "chai": "^1.9.1",  
    "mocha": "^1.20.1"  
  }  
}
```

Once you have done this, you can invoke Mocha in the following way:

```
$ me@~/validator> npm test
```

The `npm test` command will just inspect the `"scripts"` section of `package.json` and execute the command specified in `"test"`. NPM is smart enough to figure out that you want to execute a tool installed as a local package, so it will temporarily modify `PATH` to include `node_modules/.bin/`.

This is a very good approach. Not only is it very easy to execute the tests, but if we want to change the options in Mocha or use another test runner that is different from Mocha, we just need to edit the test script in package.json. This way, we will always use `npm test` and forget about the command line details of the test runner.



The popular Travis (<https://travis-ci.org/>) **Continuous Integration (CI)** platform will first execute `npm install` and then `npm test` as the default build command for a Node project.



Enough setup; it is time to see a failing test! Just create a new test file:

```
$ me@~/validator> mkdir test && touch test/validator-spec.js
```

Edit the file to add the following code:

```
var assert = require('assert');

describe('Be welcome to Mocha', function() {
  it('with a failing test', function() {
    assert(false, 'Hello World');
  });
});
```

Do not worry about the code now; simply execute `npm test` again and behold the result:

```
# iham at iham in ~/validator [18:21:47]
$ npm test

> validator@0.1.0 test /Users/iham/validator
> mocha -u bdd -R spec -t 500 --recursive

Be welcome to Mocha
  1) with a failing test

  0 passing (6ms)
  1 failing

  1) Be welcome to Mocha with a failing test:
     AssertionError: Hello World
      at Context.<anonymous> (/Users/iham/validator/test/validator-spec.js:5:5)
      at callFn (/Users/iham/validator/node_modules/mocha/lib/runnable.js:223:21)
      at Test.Runnable.run (/Users/iham/validator/node_modules/mocha/lib/runnable.js:216:7)
      at Runner.runTest (/Users/iham/validator/node_modules/mocha/lib/runner.js:373:10)
      at /Users/iham/validator/node_modules/mocha/lib/runner.js:451:12
      at next (/Users/iham/validator/node_modules/mocha/lib/runner.js:298:14)
      at /Users/iham/validator/node_modules/mocha/lib/runner.js:306:7
      at next (/Users/iham/validator/node_modules/mocha/lib/runner.js:246:23)
      at Object._onImmediate (/Users/iham/validator/node_modules/mocha/lib/runner.js:275:5)
      at processImmediate [as _immediateCallback] (timers.js:336:15)

npm ERR! Test failed. See above for more details.
npm ERR! not ok code 0
```

Your first Mocha execution

Useful options in Mocha

The Mocha **Command Line Interface (CLI)** has a multitude of options. Here, we will review the ones we used and some other important ones.



You can create the `./test/mocha.opts` file so it contains the default options for Mocha. The options specified in this file have less priority and will be replaced by the ones specified in the CLI. The file format consists of an option in a different line.

Here are the main options:

- The most important option is the `-u` or `--ui` option that allows us to specify which test interface to use to write tests. The test interface is the set of functions you will use to write your tests. The default value, `bdd`, is the one we will use throughout the book, and it indicates that we will use functions such as `describe`, `it`, `context`, `beforeEach`, and so on to organize our tests. Other test interfaces are `qunit`, `exports`, and `tdd`, but I do not find them BDD-friendly!
- Another cool option is `-R` or `--reporter`; this allows you to define the test report format you want to use. Mocha is highly configurable and allows you to write your own custom reporter. There are some open source projects that add new reporters to Mocha. We will use `spec`, which offers a very clear and detailed report. Other reports such as `dot`, `min`, or `progress` are more concise and a bit faster. Depending on the grade of detail you want to receive and your technical restrictions, you should choose one or the other. If you are in a good mood, you should try the `nyan` reporter.
- The `-t` or `--timeout` option defines how many milliseconds Mocha will wait for a test to finish. The default value of 2 seconds is too high for a rapid test-first cycle and should be used only for UI testing or end-to-end tests. In this book, we will use 500 milliseconds, which should be more than enough for most use cases.



The `--recursive` option instructs Mocha to explore the specified test folder recursively to look for tests to execute. By default, Mocha will execute the test in the files that match the `./test/*.js` pattern, but you can specify other folders, patterns, or even specific files at the end of the command line, just after all the options.

- If you want to stop the tests at the first fail, use the `-b` or `--bail` option. By default, Mocha will run all the tests, even if some of them are failing.

- Finally, the interesting `-w` or `--watch` option will execute mocha. It will not exit; instead, it will watch for changes in the current working directory; whenever a file is changed, it will re-execute all the tests. This last option, `--watch`, is very useful during development. In fact, it is so useful that we should change the package.json file as follows:

```
{  
  "name": "validator",  
  "version": "0.1.0",  
  "description": "A validation service for Weird LTD.",  
  "main": "index.js",  
  "scripts": {  
    "test": "mocha -u bdd -R spec -t 500 --recursive",  
    "watch": "mocha -u bdd -R spec -t 500 --recursive --watch"  
  },  
  "author": "Enrique Amodeo",  
  "license": "MIT",  
  "devDependencies": {  
    "chai": "^1.9.1",  
    "mocha": "^1.20.1"  
  }  
}
```

Note the new entry called `"watch"` inside the `"scripts"` section. To execute Mocha in the `watch` mode, just execute the following command:

```
$ me@~/validator> npm run-script watch
```

You can execute any command inside the `"scripts"` section, including the `test` command, issuing `npm run-script` followed by the name of the script. Since NPM always considers the `test` command as the default script, you can omit `run-script` and simply execute `npm test`. However, only the `test` script is special; for the others, you need to use `npm run-script` or, even better, its shorter alias, `npm run`.

You can see that Mocha is executed in the `watch` mode because the execution does not terminate. Mocha is just waiting for the changes to re-execute the tests, so just try to change `test/validator-spec.js` and see how Mocha reacts.

This is a really practical setup to do BDD; just change your code, and your test will be executed again, providing you with fast feedback!



You can visit the home page of Mocha at <http://mochajs.org> to get exhaustive information on the tool.

Our first test-first cycle

It is now time to start practicing a bit of test-first. For this, we will try to solve a small coding exercise, or coding kata. Do not worry if it is not very realistic, as its goal is to exercise the test-first approach and the basic usage of the tools.

Suppose you are developing a web application, and you need to write the validation logic for a field in one of the entities of the model. So, a new coding task appears to implement such a validation. According to the test-first cycle, we should first write a test, so let's open `validator-spec.js` and replace its dummy code with the following lines:

```
var assert = require('assert');

describe('A Validator', function() {
}) ;
```

This is not yet a test, but we are using the `describe` function provided by Mocha to structure what is going to be our test. The `describe` function creates a new **test suite** for Mocha. A test suite is just a grouping of test cases with a nice description. The description is provided as the first parameter, and the actual contents of the test suite are supplied inside the function that we use as the second parameter. Whenever Mocha wants to execute the test suite, it will just execute the function.

Since the description is used to generate a nice test result report, we need to have a very clear and readable one. It is a good practice to use as description the name of the unit that is actually being tested in that test suite—in this case, the validator. In this chapter, we will be doing traditional unit testing, so we should use the name of the Validator component as the description of our test suite. If we were being strict about BDD, we would have used the title of a feature as the description. However, in small systems, such as this code kata, there is no point in being so strict, as it is so small that there is a one-to-one correspondence between the feature and the component.



You can have several tests suites per file or even nested test suites.



According to the requirements of our validator, it should take a positive number, apply a set of validations, and return an array that contains all the errors. Of course, the first validation rule is really simple: if the number is not strictly positive, generate the `error.nonpositive` error.

The most simple test could be the following one:

```
var assert = require('assert');

describe('A Validator', function() {
  it('will return error.nonpositive for not strictly positive
numbers', function() {
    assert.deepEqual(validator(0), ['error.nonpositive']);
  });
});
```

First of all, note that we have created our first test case using the Mocha `it` function. Like `describe`, the `it` function takes a description of the actual test case, which will be used in the test report, and a function with the code of the test. This is a common pattern in Mocha. Unlike `describe`, the `it` function cannot be nested.

It is important that the description of the test case and the test suite can be read together as a coherent statement; this way, the reporters will offer a readable explanation of what is going on with the test.

This test case is so simple that we do not need a setup phase, and the action and assert phases are combined in a single line.

The action is simple: just call a hypothetical `validator` function with `0`. After all, we should always write the simplest test that can fail and that involves choosing simple data inputs and outputs and opting for the most simple interface for our component. If we had chosen an object instead of a function, the test would have looked like this:

```
var assert = require('assert');

describe('A Validator', function() {
  it('will return error.nonpositive for not strictly positive
numbers', function() {
    var validator = new Validator();

    assert.deepEqual(validator.validate(0), ['error.nonpositive']);
  });
});
```

Actually using an object here is more complex than using a single function, and does not really add anything!

To make the assertion, we are using a standard node package, `assert`, so we do not need to install anything extra for now. The `assert` package has a set of assertion functions, such as `equal`, `deepEqual`, and so on. They will perform the check and throw an assertion error if the result is not as expected. In this case, we are using `deepEqual` because we are comparing arrays and not simple values. The `assert.equal` function will use the `==` operator to test, but we really want to test the contents of the arrays, so we use `assert.deepEqual`.

Try to execute the test and see it fail. The error tells us that actually, there is no such thing as a `validator` function yet. This has an easy solution:

```
var assert = require('assert');

function validator() {

}

describe('A Validator', function() {
  it('will return error.nonpositive for not strictly positive numbers', function() {
    assert.deepEqual(validator(0), ['error.nonpositive']);
  });
});
```

Yes, we can write the `validator` function in the same file as the test! We are trying to fix a failing test, so we do not have time to write nice code; this is a task for when the test passes. If you execute the test, you will see a proper fail now, which will look something like this:

```
# iham at iham in ~/validator [20:42:50]
$ npm test

> validator@0.1.0 test /Users/iham/validator
> mocha -u bdd -R spec -t 500 --recursive

A Validator
  1) will return error.nonpositive for not strictly positive numbers

  0 passing (6ms)
  1 failing

  1) A Validator will return error.nonpositive for not strictly positive numbers:
     AssertionError: "undefined" deepEqual [ 'error.nonpositive' ]
      at Context.<anonymous> (/Users/iham/validator/test/validator-spec.js:9:12)
      at callFn (/Users/iham/validator/node_modules/mocha/lib/runnable.js:223:21)
      at Test.Runnable.run (/Users/iham/validator/node_modules/mocha/lib/runnable.js:216:7)
      at Runner.runTest (/Users/iham/validator/node_modules/mocha/lib/runner.js:373:10)
      at /Users/iham/validator/node_modules/mocha/lib/runner.js:451:12
      at next (/Users/iham/validator/node_modules/mocha/lib/runner.js:298:14)
      at /Users/iham/validator/node_modules/mocha/lib/runner.js:308:7
      at next (/Users/iham/validator/node_modules/mocha/lib/runner.js:246:23)
      at Object._onImmediate (/Users/iham/validator/node_modules/mocha/lib/runner.js:275:5)
      at processImmediate [as _immediateCallback] (timers.js:336:15)

npm ERR! Test failed. See above for more details.
npm ERR! not ok code 0
```

Our first real error

We should make the test pass now; let's do it with the simplest code possible:

```
var assert = require('assert');

function validator() {
    return ['error.nonpositive'];
}

describe('A Validator', function() {
    it('will return error.nonpositive for not strictly positive
numbers', function() {
        assert.deepEqual(validator(0), ['error.nonpositive']);
    });
});
```

Now, it works! Yes, the code is not impressive, but actually you cannot do it better. All the tests pass, and the implementation is so simple that you cannot argue against it. Of course, what is happening is that we only have one test; if we add more tests, the code will grow more complex.

Now, with all the tests passing, we can clean our code a bit. The obvious thing is to extract the production code to another file. Create a `lib/validator.js` file and move the `validator` function to it:

```
module.exports = function () {
    return ['error.nonpositive'];
};
```

Now, edit the test code to get the following lines:

```
var assert = require('assert'),
    validator = require('../lib/validator');

describe('A Validator', function() {
    it('will return error.nonpositive for not strictly positive
numbers', function() {
        assert.deepEqual(validator(0), ['error.nonpositive']);
    });
});
```

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.



Then, just run the tests again to make sure that everything is OK, as follows:

```
# iham at iham in ~/validator [21:08:12]
$ npm test

> validator@0.1.0 test /Users/iham/validator
> mocha -u bdd -R spec -t 500 --recursive

A Validator
  ✓ will return error.nonpositive for not strictly positive numbers

1 passing (6ms)
```

Green and refactored

This concludes our very first test-first iteration, but there are still several more! However, before going ahead, let's have a look at Chai.

More expressive assertions with Chai

We have already installed `chai` as a dependency of our project, but we have not yet used it. Specifically, I am using version 1.9.1, but any 1.x version should be OK.

Until now, we have been using the standard `assert` package to create our assertions. The `assert` package is not bad, but it is limited to a few assertions. It is not extensible, and some people found it a bit difficult to read. I personally always find myself wondering, "Is the first parameter the actual value or the expected one?" Actually, the first parameter is the actual value, and the second one is the expected value. This is important because the report message depends on distinguishing the actual value from the expected one. Let's change the code of our test to use `chai`:

```
var chai = require('chai'),
  expect = chai.expect,
  validator = require('../lib/validator');

describe('A Validator', function() {
  it('will return error.nonpositive for not strictly positive numbers', function() {
    expect(validator(0)).to.be.deep.equal(['error.nonpositive']);
  });
});
```

As you see, now we are importing the `chai` package instead of the `assert` one. Note that `chai` is just a library and not a test runner such as Mocha, so we do not need to change the test script inside the `package.json` file.

Using the imported `chai` object, we created a local variable called `expect`; this points to the `expect` utility function in `chai`. This way, we simply need to type `expect` in our code instead of `chai.expect`.

In the test itself, we can see that we wrap the actual result of `validator()` using the `expect` function. This function will return a nice DSL that we can use to write nice and expressive assertions about the actual result. This DSL is formed by particles that can be chained together. There are three kinds of particles: chains, assertions, and flags. They are explained here:

- **Chains** are particles that do not modify the behavior of the assertions but that provide expressivity. You can always add a `".` to a chain to add another particle. In this example, we used the chains `to` and `be`, but we could have omitted them, and the test would have been exactly the same. Chains make our assertions easier to read. The `to`, `be`, `been`, `is`, `that`, `and`, `has`, `have`, `with`, `at`, `of`, and `same` particles are considered to be chains.
- **Assertions** are the particles that perform the actual check of the result. They are usually functions that take one or more parameter with the expected result. In the example, we are using the `equal` assertion.
- **Flags** allow us to modify the behavior of the assertions. For example, the `equal` assertion just checks the value using the `==` operator. However, if we use the `deep` flag, as we did in the example, the `equal` assertion will instead check the contents of the actual result. Another useful flag is `not`; this will invert the assertion.

Other useful assertions are `include` or `contain`; both are the same. These assertions simply test whether a string contains a substring or an array of elements, as in the following examples:

```
expect(['a', 5, 'cd']).to.contain(5);
expect('string').to.contain('tri');
```

To perform a Boolean assertion, we can use the `ok`, `true`, or `false` particles. These assertions are not functions. For example, take a look at the following code:

```
expect(true).to.be.ok;
expect(1).to.be.ok;
expect(true).to.be.true;
expect(1).not.to.be.true;
expect('').not.to.be.ok;
expect(false).to.be.false;
```

The `ok` assertion will test whether the actual result is a *truthy* value, whereas `true` and `false` will test for strict equality to the `true` and `false` primitives in JavaScript. We also can test strictly for `null` and `undefined` using the corresponding assertions. If we just want to know whether a value is either `null` or `undefined`, we could use the `exist` assertion:

```
expect(!true).to.be.false;
expect(!false).to.be.true;
expect(false).to.exist;
expect(null).not.to.exist;
expect(undefined).not.to.exist;
```

If you wish to check for the type of the actual result, you could use the `a` or an `an` assertion. In `chai`, the type is the result of the JavaScript `typeof` operator, except for `null`. Unfortunately, in JavaScript, `typeof null` will return '`object`' instead of '`null`', which does not make sense at all. Fortunately `chai` takes care of this annoying detail:

```
expect(null).to.be.a('null'); // Thanks Chai!
expect(null).to.be.null; // Of course
expect(undefined).to.be.an('undefined');
expect(undefined).to.be.undefined;
expect(Number).to.be.an('object');
expect(Number).to.be.a('function');
expect(true).to.be.a('boolean');
expect(3).to.be.a('number');
expect('John').to.be.a('string');
```

For arrays, objects, and strings, we can use the `empty` assertion:

```
expect('').to.be.empty;
expect({}).to.be.empty;
expect([]).to.be.empty;
```

To check numbers, we have several assertions, such as `most`, `least`, `above`, `below`, `closeTo`, and `within`. Let's see some examples:

```
expect(1).to.be.at.most(2);
expect(2).to.be.at.most(2);
expect(3).not.to.be.at.most(2);
expect(1).to.be.below(2);
expect(2).not.to.be.below(2);
expect(1).to.be.below(2);
expect(1).to.be.at.least(1);
expect(2).to.be.at.least(1);
```

```
expect(0).not.to.be.at.least(1);
expect(1).not.to.be.above(1);
expect(2).to.be.above(1);
expect(1).to.be.within(1, 3);
expect(2).to.be.within(1, 3);
expect(3).to.be.within(1, 3);
expect(4).not.to.be.within(1, 3);
expect(2.2).to.be.closeTo(2, 0.2);
expect(2.3).not.to.be.closeTo(2, 0.2);
```

All these assertions have aliases such as `gt`, `greaterThan`, `gte`, `lt`, `lessThan`, and `lte`.

Sometimes, you would like to test whether a function throws an error. You can do this with the `throw` assertion or with its alias, `throws`. For example, if we expect the validator to throw when a null value is passed, we can test as in the following code:

```
var fn = function() {
  validator(null);
};

expect(fn).to.throw('parameter');
expect(fn).to.throw(ValidatorError);
expect(fn).to.throw(new ValidatorError('Null is bad parameter'));
expect(fn).to.throw(/bad parameter/);
```

We can test for a specific error type passing the constructor of the error, for the message using a string or a regular expression, or test whether the error is strictly equal to the one we provide.

Some particles, such as `include` and `contain`, can act as a flag or assertion. For example, both `include` and `contain` can be used as a flag in the following example:

```
expect({ name: 'John', age: 32 }).to.include.keys('age');
```

This will test whether the object contains a key. Without `include` or `contain`, the `keys` assertion would have failed, as it would have checked whether the object only has the property `'age'`.

Another particle that can work as an assertion or a flag is `length`:

```
expect([1, 2, 3]).to.have.length(3);
expect([1, 2, 3]).to.have.length.of.at.least(2);
expect([1, 2, 3]).to.have.length.of.at.most(4);
expect([1, 2, 3]).to.have.length.within(2, 4);
expect([1, 2, 3]).to.have.length.greaterThan(2);
```

As you can see, as an assertion we can use it to test whether the length of an array has a specific value. Alternatively, we can use it as a flag and combine it with any other assertion that works with numbers.



There are more assertions and flags in Chai; for an exhaustive reference of them, go to <http://chaijs.com/api/bdd/>.



We can merge several assertions in one as if we are using the `and` chain. For example, assume the following two assertions:

```
expect(anArray).to.have.length(2);  
expect(anArray).to.contain('element');
```

We can merge them in a single statement:

```
expect(anArray).to.have.length(2).and.to.contain('element');
```

This nice trick can help us, sometimes, to make more expressive assertions.

Working with the "should" interface

Chai provides three different kinds of assertion styles: `expect`, `should`, and `assert`. The `assert` style is very similar to the one provided in the standard `assert` package, but with more functionality. We have already seen the `expect` style, so let's have a look at the `should` style.



In this book, we will not use the `assert` style. If you are really interested, have a look at <http://chaijs.com/api/assert/>.



To use the `should` style, we can change the test as follows:

```
var chai = require('chai'),  
    should = chai.should(),  
    validator = require('../lib/validator');  
  
describe('A Validator', function() {  
  it('will return error.nonpositive for not strictly positive  
  numbers', function() {  
    validator(0).should.be.deep.equal(['error.nonpositive']);  
  });  
});
```

The first change you can notice is that we need to call a function, `chai.should()`, to get a reference to the `should` utility, but we are actually not referencing it in our test!

What happens is that `should` will install a set of extensions to the `Object` prototype; in other words, it will monkey-patch the global prototype for all the objects in JavaScript. This way, we do not need to wrap the actual result we want to check with any utility function. We can directly use the DSL to make a nice assertion chain!

In general, whenever we see code such as `expect(expr).to.`, we can replace it with `expr.should.` using the `should` style. The assertions, flags, and chains are exactly the same as the ones we saw in the `expect` style.

This is a bit more elegant than the `expect` approach of wrapping the result with the `expect` function. However, some people are not very comfortable about overriding the `Object` prototype, as it can lead to some problems in Internet Explorer. Anyway, in Node, this will not be a problem.

However, the `should` approach has another problem: how do we test for `null` or `undefined`? What will happen if the actual result is `null` or `undefined` itself? In this case, the `should` approach does not work, and we need to use some workarounds. The problem is that `null` and `undefined` cannot be extended with the Chai DSL, as they are not really true objects. So, the only thing we will see in this case is a nasty exception.

The `should` utility object contains a set of functions that we can use in these cases. For example, to test for `null` or `undefined`, we can do the following:

```
should.exist({});  
should.not.exist(null);  
should.not.exist(undefined);
```

If the code we are testing returns `null` or `undefined`, then we can use the `should.equal` utility that works as if it is in the `assert` style:

```
should.equal(maybeNull(), 'but must be non null');
```

However, in this case, we lose all the power of Chai! This is a big problem in my opinion, since, actually, almost any implementation could return `null` or `undefined`, intentionally or just because of a mistake. That is why the `should` style is not a good idea in JavaScript; it is better to use the `expect` style itself. The `should` style is a good idea in languages such as Ruby, where `nil` is a real object, but not in JavaScript!

Red/Green/Refactor

Now that we have a basic knowledge of Chai, we can continue with our coding. The code we already have in place is clearly not correct, so we need to add another test that forces us to make the code right.

Currently, we are implementing the rule about errors for nonpositive numbers, so we should finish it before continuing. Which test could we add to make our implementation fail? We can try testing the validator with valid numbers.

Let's see the result:

```
describe('A Validator', function() {
  it('will return no errors for valid numbers', function() {
    expect(validator(3)).to.be.empty;
  });

  it('will return error.nonpositive for not strictly positive
numbers', function() {
    expect(validator(0)).to.be.deep.equal(['error.nonpositive']);
  });
});
```

Note that we have added a new test for valid numbers that is failing. So now we fix it in a very simple way in `lib/validator.js`:

```
module.exports = function (n) {
  if(n === 0)
    return ['error.nonpositive'];
  return [];
};
```

The test passes, but the code still seems to be incorrect. Fortunately, we can break it easily by adding another test for nonstrictly positive numbers:

```
describe('A Validator', function() {
  it('will return no errors for valid numbers', function() {
    expect(validator(3)).to.be.empty;
  });

  it('will return error.nonpositive for not strictly positive numbers,
like 0', function() {
    expect(validator(0)).to.be.deep.equal(['error.nonpositive']);
  });
});
```

```
it('will return error.nonpositive for not strictly positive numbers, like -2', function() {
  expect(validation(-2)).to.be.deep.equal(['error.nonpositive']);
});
```

Note that the new test has exactly the same description as the other one; after all, they are testing the same rule. The only difference is that we are testing with different inputs, so we changed the descriptions to identify the specific input we are using in each test.

To fix the test, we only need to make a very small change:

```
module.exports = function (n) {
  if(n === 0 || n === -2)
    return ['error.nonpositive'];
  return [];
};
```

Now, we have all tests green, and it is time to clean the code. We have a duplication in the `if` condition with the same equality check twice. We also have duplication between the literals in the test, `0` and `-2`, and the literals used in the `validator` function. It is time to remove this duplication collapsing both checks:

```
module.exports = function (n) {
  if(n <= 0)
    return ['error.nonpositive'];
  return [];
};
```

Yes, I know that you knew the correct code from the beginning! However, this is only because the logic we are implementing in this exercise is simple and very evident. In a real scenario, you will not have such simple problems. The technique of adding several tests for the same logic but using different inputs in order to expose some duplication that drives our refactor is called **triangulation**. The different inputs that we use during triangulation are called **examples**. We should choose examples that are meaningful from the point of view of the problem domain, that break our current implementation, and that try to cover edge cases. When the logic is simple, a couple of examples are enough but, when it is not so simple, you need more to discover the right refactor or the hidden algorithm. The point of triangulation is that it will generate a lot of duplication and ugly code; sooner or later, though, you will discover a pattern to collapse all this duplication or make your code cleaner.

So, refactored and green? Not quite yet. We still need to clean our tests! There is some duplication in the test case's description. We can solve this with nested test suites:

```
describe('A Validator', function() {
  it('will return no errors for valid numbers', function() {
    expect(validator(3)).to.be.empty;
  });

  describe('will return error.nonpositive for not strictly positive numbers:', function() {
    it('like 0', function() {
      expect(validator(0)).to.be.deep.equal(['error.nonpositive']);
    });

    it('like -2', function() {
      expect(validator(-2)).to.be.deep.equal(['error.nonpositive']);
    });
  });
});
```

We used a nested test suite to describe one of the validation rules of the `validator` function. Inside this nested suite, we added one test per example. Since we still have more rules to implement, there will be more nested tests suites. Let's continue with the next rule: numbers divisible by 3 generate `error.three`. However, this time, we will do it faster:

```
describe('A Validator', function() {
  it('will return no errors for valid numbers', function() {
    expect(validator(7)).to.be.empty;
  });
  // Skipped code for brevity
  describe('will return error.three for divisible by 3 numbers:', function() {
    it('like 3', function() {
      expect(validator(3)).to.be.deep.equal(['error.three']);
    });

    it('like 6', function() {
      expect(validator(6)).to.be.deep.equal(['error.three']);
    });
  });
});
```

Note that now there is a new nested suite to test the new rule. Also note that we modified the test about valid numbers, as the example we had selected before, 3, is not a valid number after all. The implementation to make the test pass is as follows:

```
module.exports = function (n) {
  if(n <= 0)
    return ['error.nonpositive'];
  if(n % 3 === 0)
    return ['error.three'];
  return [];
};
```

Now the test is green, but we can change the code to have only one return value and one array instantiation:

```
module.exports = function (n) {
  var result = [];
  if(n <= 0)
    result.push('error.nonpositive');
  else if(n % 3 === 0)
    result.push('error.three');
  return result;
};
```

The next validation rule tells us that numbers divisible by 5 are also invalid! It is time for another suite:

```
describe('A Validator', function() {
  it('will return no errors for valid numbers', function() {
    expect(validator(7)).to.be.empty;
  });

  // Skipped code for brevity
  describe('will return error.five for divisible by 5 numbers:', function() {
    it('like 5', function() {
      expect(validator(5)).to.be.deep.equal(['error.five']);
    });
  });
});
```

```
it('like 10', function() {
  expect(validator(10)).to.be.deep.equal(['error.five']);
});
});
});
});
```

To implement this, we can simply do a bit of copy-and-paste:

```
module.exports = function (n) {
  var result = [];
  if(n <= 0)
    result.push('error.nonpositive');
  else if(n % 3 === 0)
    result.push('error.three');
  else if(n % 5 === 0)
    result.push('error.five');
  return result;
};
```

Now, we have the tests passing again. Fortunately, there are no more validation rules to implement. However, we are not yet done, since we can still think of a failing test. What should happen if there is a number that violates several rules? In this case, there should be an error for each of the violated rules in the array. In fact, this is what happened when the QA guy made some exploratory testing and introduced the number 15 in our app. What a shame!

No problem, we can add this bug in our suite:

```
var chai = require('chai'),
  expect = chai.expect,
  validator = require('../lib/validator');

describe('A Validator', function() {
  // Skipped code for brevity
  it('will return one error for each rule the number violates',
  function() {
    expect(validator(15)).to.be.deep.equal(['error.three', 'error.
five']);
  });
});
```

If you run the tests now, you should reproduce the bug:

```
A Validator
  ✓ will return no errors for valid numbers
  1) will return one error for each rule the number violates
    will return error.nonpositive for not strictly positive numbers:
      ✓ like 0
      ✓ like -2
    will return error.three for divisible by 3 numbers:
      ✓ like 3
      ✓ like 6
    will return error.five for divisible by 5 numbers:
      ✓ like 5
      ✓ like 10

7 passing (12ms)
1 failing

1) A Validator will return one error for each rule the number violates:
   AssertionError: expected [ 'error.three' ] to deeply equal [ 'error.three', 'error.five' ]
+   expected - actual
[  
+   "error.three",
+   "error.five"
-   "error.three"
]  
  
at Assertion.deepEqual (/Users/iham/validator/node_modules/chai/lib/chai/core/assertions.js:11:11)
at Assertion.ctx.(anonymous function) [as equal] (/Users/iham/validator/node_modules/chai/lib/chai/core/assertions.js:11:11)
at Context.<anonymous> (/Users/iham/validator/test/validator-spec.js:41:38)
```

Gotcha! The error is captured

Note the nice report we have in the screenshot. It nicely displays the difference between the expected (in green) and the actual result (in red). Whenever there is an assertion fail, Mocha expects `AssertionError` to be thrown. This error has three important fields: `actual`, `expected`, and `showDiff`. If `showDiff` is set to `true`, then the reporter is supposed to display the actual difference between the actual and expected result. You can tell Chai not to activate this flag, with the `chai.config.showDiff = false;` instruction.

We could have written the test in another way; instead of adding a new test case, we can modify the existing test cases to reflect the new behavior:

```
var chai = require('chai'),
  expect = chai.expect,
  validator = require('../lib/validator');

describe('A Validator', function() {
```

```
it('will return no errors for valid numbers', function() {
    expect(validator(7)).to.be.empty;
});

describe('will include error.nonpositive for not strictly positive numbers:', function() {
    it('like 0', function() {
        expect(validator(0)).to.include('error.nonpositive');
    });
    it('like -2', function() {
        expect(validator(-2)).to.include('error.nonpositive');
    });
});

describe('will include error.three for divisible by 3 numbers:', function() {
    it('like 3', function() {
        expect(validator(3)).to.include('error.three');
    });
    it('like 15', function() {
        expect(validator(15)).to.include('error.three');
    });
});

describe('will include error.five for divisible by 5 numbers:', function() {
    it('like 5', function() {
        expect(validator(5)).to.include('error.five');
    });
    it('like 15', function() {
        expect(validator(15)).to.include('error.five');
    });
});
```

There are some changes. The first change is to test using `include` instead of strict equality, and we changed the descriptions of the tests accordingly. The second change is that we replaced `6` and `10` with `15` in the tests. This approach is better, as these new tests reflect more effectively how the validator should work. Furthermore, this approach helps us write less tests!

To fix the bug, we will simply remove the `else` keyword:

```
module.exports = function (n) {
  var result = [];
  if(n <= 0)
    result.push('error.nonpositive');
  if(n % 3 === 0)
    result.push('error.three');
  if(n % 5 === 0)
    result.push('error.five');
  return result;
};
```

Now, the test is passing without any bugs. However, with this last change, a pattern emerged: we can extract each `if` condition to a rule function:

```
function nonPositiveValidationRule(n, result) {
  if (n <= 0)
    result.push('error.nonpositive');
}
function nonDivisibleBy3ValidationRule(n, result) {
  if (n % 3 === 0)
    result.push('error.three');
}
function nonDivisibleBy5ValidationRule(n, result) {
  if (n % 5 === 0)
    result.push('error.five');
}

module.exports = function (n) {
  var result = [];
  nonPositiveValidationRule(n, result);
  nonDivisibleBy3ValidationRule(n, result);
  nonDivisibleBy5ValidationRule(n, result);
  return result;
};
```

Now we have an explicit representation of the validation rules in our code. However, we can simplify it even more; `nonDivisibleBy3ValidationRule` and `nonDivisibleBy5ValidationRule` have a clear duplication:

```
function nonPositiveValidationRule(n, result) {
  if (n <= 0)
    result.push('error.nonpositive');
}
```

```
function makeNonDivisibleValidationRule(divisor, error) {
    return function(n, result) {
        if (n % divisor === 0)
            result.push(error);
    };
}
var nonDivisibleBy3ValidationRule = makeNonDivisibleValidationRule(3,
'error.three'),
    nonDivisibleBy5ValidationRule = makeNonDivisibleValidationRule(5,
'error.five');

module.exports = function (n) {
    var result = [];
    nonPositiveValidationRule(n, result);
    nonDivisibleBy3ValidationRule(n, result);
    nonDivisibleBy5ValidationRule(n, result);
    return result;
};
```

Now we can avoid calling three functions in a row using a reduce loop:

```
function nonPositiveValidationRule(n, result) {
    if (n <= 0)
        result.push('error.nonpositive');
}
function makeNonDivisibleValidationRule(divisor, error) {
    return function(n, result) {
        if (n % divisor === 0)
            result.push(error);
    };
}
var validationRules = [
    nonPositiveValidationRule,
    makeNonDivisibleValidationRule(3, 'error.three'),
    makeNonDivisibleValidationRule(5, 'error.five')
];
module.exports = function (n) {
    return validationRules.reduce(function(result, rule) {
        rule(n, result);
        return result;
    }, []);
};
```

Now we can extract each rule to a different file. We move the rule about nonpositive numbers to `lib/validator/rules/nonPositive.js`:

```
module.exports = function (n, result) {
  if (n <= 0)
    result.push('error.nonpositive');
};
```

The other rule goes to the `lib/validator/rules/nonDivisible.js` file, as follows:

```
module.exports = function (divisor, error) {
  return function (n, result) {
    if (n % divisor === 0)
      result.push(error);
  };
};
```

Finally, we can use these files inside `lib/validator.js`:

```
var nonPositiveValidationRule = require('./rules/nonPositive'),
nonDivisibleValidationRule = require('./rules/nonDivisible'),
validationRules = [
  nonPositiveValidationRule,
  nonDivisibleValidationRule(3, 'error.three'),
  nonDivisibleValidationRule(5, 'error.five')
];
module.exports = function (n) {
  return validationRules.reduce(function (result, rule) {
    rule(n, result);
    return result;
  }, []);
};
```

We cannot think of another failing test or refactor, so we are done!

Parameterized tests

The tests are currently OK, but sometimes we end up testing the same thing again and again, but with different examples. In fact, you can see a lot of duplication of code in our current tests. We can remove them using **parameterized tests** that we can execute several times with different examples:

```
function expectedToIncludeErrorWhenInvalid(number, error) {
  it('like ' + number, function () {
    expect(validator(number)).to.include(error);
```

```
    });
}

describe('A Validator', function () {
  it('will return no errors for valid numbers', function () {
    expect(validator(7)).to.be.empty;
  });

  describe('will include error.nonpositive for not strictly positive numbers:', function () {
    expectedToIncludeErrorWhenInvalid(0, 'error.nonpositive');
    expectedToIncludeErrorWhenInvalid(-2, 'error.nonpositive');
  });

  describe('will include error.three for divisible by 3 numbers:', function () {
    expectedToIncludeErrorWhenInvalid(3, 'error.three');
    expectedToIncludeErrorWhenInvalid(15, 'error.three');
  });

  describe('will include error.five for divisible by 5 numbers:', function () {
    expectedToIncludeErrorWhenInvalid(5, 'error.five');
    expectedToIncludeErrorWhenInvalid(15, 'error.five');
  });
});
```

We can make this technique as complex as we want, for example by using a loop:

```
function expectedToIncludeErrorWhenInvalid(example) {
  var number = example.number,
      error = example.error;
  it('like ' + number, function () {
    expect(validator(number)).to.include(error);
  });
}

describe('A Validator', function () {
  it('will return no errors for valid numbers', function () {
    expect(validator(7)).to.be.empty;
  });

  describe('will include error.nonpositive for not strictly positive numbers:', function () {
    [
```

```
{number: 0, error: 'error.nonpositive'},
{number: -2, error: 'error.nonpositive'}
].forEach(expectedToIncludeErrorWhenInvalid);
});

describe('will include error.three for divisible by 3 numbers:', function () {
[
  {number: 3, error: 'error.three'},
  {number: 15, error: 'error.three'}
].forEach(expectedToIncludeErrorWhenInvalid);
});

describe('will include error.five for divisible by 5 numbers:', function () {
[
  {number: 5, error: 'error.five'},
  {number: 15, error: 'error.five'}
].forEach(expectedToIncludeErrorWhenInvalid);
});
});
```

Of course, in our use case it is simply over-engineering. It could pay off if you really have plenty of examples, but do not abuse this technique.



We can use a parameterized test to refactor a test suite that must be tested against several implementations of the same interface. In this case, we can extract the common test suite to a different file and import it from the specific tests suites of each implementation.

Organizing your setup

So far, we have a validator with a fixed set of validation rules. We can reuse the validator in different contexts if we can make the set of rules a configuration parameter. In fact, in our imaginary requisite list, it is stated that the set of rules can be changed using some kind of configuration system!

This time, we need to change the interface of the validator package a bit to include the possibility of configuring the rules. To do so, we will change our test to reflect the new interface. We need a *setup* phase where we create a validator instance with the desired set of rules:

```
var chai = require('chai'),
expect = chai.expect,
validatorWith = require('../lib/validator'),
```

```
nonPositiveValidationRule = require('../lib/rules/nonPositive'),
nonDivisibleValidationRule = require('../lib/rules/nonDivisible');

describe('A Validator', function() {
  it('will return no errors for valid numbers', function() {
    var validator = validatorWith([
      nonPositiveValidationRule,
      nonDivisibleValidationRule(3, 'error.three'),
      nonDivisibleValidationRule(5, 'error.five')
    ]);
    expect(validator(7)).to.be.empty;
  });
}

describe('will include error.nonpositive for not strictly positive numbers:', function() {
  it('like 0', function() {
    var validator = validatorWith([
      nonPositiveValidationRule,
      nonDivisibleValidationRule(3, 'error.three'),
      nonDivisibleValidationRule(5, 'error.five')
    ]);
    expect(validator(0)).to.include('error.nonpositive');
  });

  it('like -2', function() {
    var validator = validatorWith([
      nonPositiveValidationRule,
      nonDivisibleValidationRule(3, 'error.three'),
      nonDivisibleValidationRule(5, 'error.five')
    ]);
    expect(validator(-2)).to.include('error.nonpositive');
  });
});

describe('will include error.three for divisible by 3 numbers:', function() {
  it('like 3', function() {
    var validator = validatorWith([
      nonPositiveValidationRule,
      nonDivisibleValidationRule(3, 'error.three'),
      nonDivisibleValidationRule(5, 'error.five')
    ]);
  });
});
```

```
    expect.validator(3)).to.include('error.three');
});

it('like 15', function() {
  var validator = validatorWith([
    nonPositiveValidationRule,
    nonDivisibleValidationRule(3, 'error.three'),
    nonDivisibleValidationRule(5, 'error.five')
  ]);
  expect.validator(15)).to.include('error.three');
});

describe('will include error.five for divisible by 5 numbers:', function() {
  it('like 5', function() {
    var validator = validatorWith([
      nonPositiveValidationRule,
      nonDivisibleValidationRule(3, 'error.three'),
      nonDivisibleValidationRule(5, 'error.five')
    ]);
    expect.validator(5)).to.include('error.five');
  });

  it('like 15', function() {
    var validator = validatorWith([
      nonPositiveValidationRule,
      nonDivisibleValidationRule(3, 'error.three'),
      nonDivisibleValidationRule(5, 'error.five')
    ]);
    expect.validator(15)).to.include('error.five');
  });
});
});
```

The tests are now broken, since we need to change the current implementation of the package to the new interface:

```
module.exports = function (validationRules) {
  return function (n) {
    return validationRules.reduce(function (result, rule) {
      rule(n, result);
      return result;
    }, []);
  };
};
```

Now the tests are passing. However, they are really ugly with a lot of duplication. We can clean them using the `beforeEach` function from Mocha:

```
describe('A Validator', function() {
  var validator;
  beforeEach(function() {
    validator = validatorWith([
      nonPositiveValidationRule,
      nonDivisibleValidationRule(3, 'error.three'),
      nonDivisibleValidationRule(5, 'error.five')
    ]);
  });

  it('will return no errors for valid numbers', function() {
    expect(validator(7)).to.be.empty;
  });

  describe('will include error.nonpositive for not strictly positive numbers:', function() {
    it('like 0', function() {
      expect(validator(0)).to.include('error.nonpositive');
    });

    it('like -2', function() {
      expect(validator(-2)).to.include('error.nonpositive');
    });
  });

  describe('will include error.three for divisible by 3 numbers:', function() {
    it('like 3', function() {
      expect(validator(3)).to.include('error.three');
    });

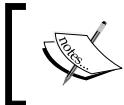
    it('like 15', function() {
      expect(validator(15)).to.include('error.three');
    });
  });

  describe('will include error.five for divisible by 5 numbers:', function() {
    it('like 5', function() {
```

```
    expect.validator(5).to.include('error.five');
});

it('like 15', function() {
  expect.validator(15).to.include('error.five');
});
});
});
});
```

The `beforeEach` function is executed once before each `it` function inside a given `describe` scope. This will generate a brand new `validator` instance for each one of the tests. This is good because we do not want the tests to interfere with each other; we want to isolate them. Executing each test with a brand new instance is a good practice, since one test can change the internal state of the object under test.



If you are an advanced JavaScript practitioner, you know that the `validator` instance is not a regular object but a **closure**. However, this does not change the point of the discussion.

Actually, our `validator` instance is immutable and stateless, so we can share the same instance across tests safely. We can leverage the `before` function in Mocha to do so:

```
describe('A Validator', function() {
  var validator;
  before(function() {
    validator = validatorWith([
      nonPositiveValidationRule,
      nonDivisibleValidationRule(3, 'error.three'),
      nonDivisibleValidationRule(5, 'error.five')
    ]);
  });
  // Skipped for brevity
});
```

The `before` function will execute exactly once before any test. This way, it will create a single instance of `validator` that will be shared by the entire test. The `before` function is also useful when you want to do something expensive only once, such as setting up a database, starting a server, or opening a WebDriver session.



Use `beforeEach` by default; it will save you a lot of headaches. Use `before` only when you want to emphasize the fact that the object you are testing is immutable or when you want to perform a one-time expensive setup.

Sometimes, you need to do some postprocessing or cleaning up after the tests. You can use the `afterEach` and `after` functions. The `afterEach` function will be called once after each test is finished. The `after` function will be called once when all the tests are finished. Both functions will be called even if some tests fail or throw an unexpected exception.

Defining test scenarios

It is obvious that the tests we have written are only OK for the set of rules we have configured for the validator. If we had configured a different set of rules, the tests would be different. In this case, it is not a big deal, since we only have one set of rules. However, it is probable that, in the future, we will have several rules, so it is better to be explicit. The easiest way to do so is to have a different test suite file for each set of rules and change the title of the test suite:

```
describe('A Validator with the default validation rules', function() {
  // Skipped for brevity
});
```

However, having all the different setups in the same test suite is also a good idea. For this, we can use the `context` function:

```
describe('A Validator', function () {
  var validator;
  context('with the default validation rules', function () {
    beforeEach(function () {
      validator = validatorWith([
        nonPositiveValidationRule,
        nonDivisibleValidationRule(3, 'error.three'),
        nonDivisibleValidationRule(5, 'error.five')
      ]);
    });

    it('will return no errors for valid numbers', function () {
      expect(validator(7)).to.be.empty;
    });
    // Skipped for brevity
  });
});
```

```
context('with other rules', function() {
  beforeEach(function() {
    validator = validatorWith([weirdRule(1), nonStandardRule]);
  });
  // Other tests
});
});
```

We can see that, for each `context`, we have at least one `beforeEach` block. We use the context to group all the tests that need a common setup and to add a nice common description to this set of tests. The `context` title should reflect the things that change between different setups—in this case, the different set of rules we have configured to the validator.

As a rule of thumb, we can consider that we can use one `context` for each scenario of a feature. A **scenario** defines a different execution path of the same feature. Since one feature defines only one operation on the system, different scenarios can vary only in their setup or in the inputs of the operation. From a technical point of view, `context` is a simple alias of `describe`. They are exactly the same as far as Mocha is concerned; they can be nested and mixed without any problem.

As a rule of thumb, it is better to reserve `context` to define scenarios, `describe` to define features and actions, and `it` for assertions or tests. If we are very purist about this, we need to change the descriptions in our test a bit:

```
describe('A validation', function () {
  var validator;
  context('using the default validation rules:', function () {
    // Skipped for brevity
    it('for valid numbers, will return no errors', function () {
      expect(validator(7)).to.be.empty;
    });

    context('for not strictly positive numbers:', function () {
      it('like 0, will include error.nonpositive', function () {
        expect(validator(0)).to.include('error.nonpositive');
      });

      it('like -2, will include error.nonpositive', function () {
        expect(validator(-2)).to.include('error.nonpositive');
      });
    });
    context('for numbers divisible by 3:', function () {
      it('like 3, will include error.three', function () {
        expect(validator(3)).to.include('error.three');
      });
    });
  });
});
```

```
});

it('like 15, will include error.three', function () {
  expect(validator(15)).to.include('error.three');
});

context('for numbers divisible by 5:', function () {
  it('like 5, will include error.five', function () {
    expect(validator(5)).to.include('error.five');
  });

  it('like 15, will include error.five', function () {
    expect(validator(15)).to.include('error.five');
  });
});
});
```

Note that, with the top-level describe function, we used the feature as the title, instead of the component name. The inner level describe functions have been replaced by context functions, as we are describing different scenarios (what happens with different inputs). For the it functions, we use the specific input example and the expected result or assertion.

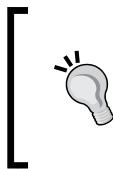
Although it is not convenient to overdo it, it is good to try to have a common naming standard across the whole team for the titles and the way you structure your Mocha tests.

Test doubles with Sinon

A few weeks after going to production, our validator is a complete success but it also happens that potential new customers would like to have different sets of validation rules. Fortunately, our validator is configurable, but we somehow need to specify a different configuration for each customer.

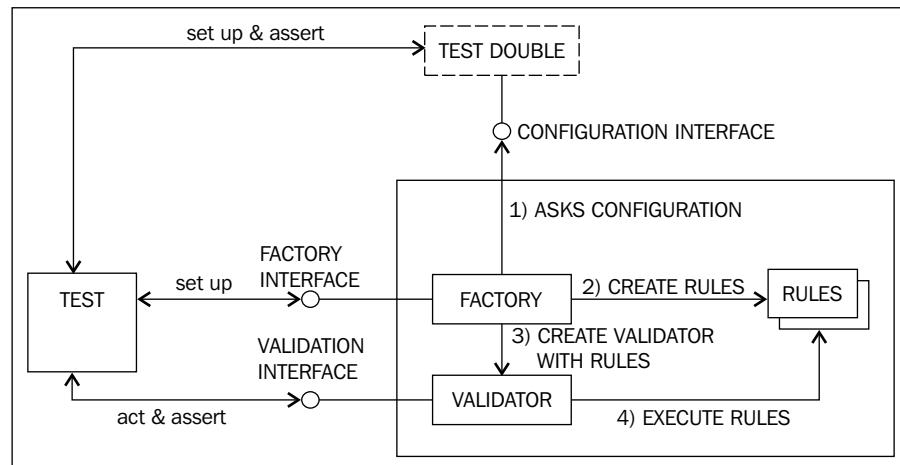
The act of configuring a different set of rules for each customer is clearly another operation of the system and, hence, a different feature, so we are not really interested in testing it in the validator tests. We would like to isolate the tests about the validator feature and the configuration feature. To do so, we need to create an interface in a way that allows the validator to ask for the correct set of rules for the configuration. Then we can create a **test double** for such an interface in our tests. The test double will impersonate the real configuration and allow us to isolate our tests.

When creating these test doubles we need to design the shape of the interfaces and how both features collaborate between each other using this interface. For the sake of being brief, let's suppose that the validator asks the configuration for a rule set using a name, and it receives JSON with a description of the rule set.



We could have thought of making the configuration return the actual instances of the validation rules. However, this would imply that the validation rules are owned by the configuration feature but this seems wrong. After all, the configuration is about configuring the system, and the validation logic should be inside the validator.

Our validator package is now more of a validator factory that uses the configuration to create a given validator instance for a set of rules. The design is as follows:



Our new design

With this in mind, we need to change the tests to reflect this new design:

```

var chai = require('chai'),
expect = chai.expect,
factoryWithConfiguration = require('../lib/factory');

describe('A validation', function () {
  var validator;
  context('using the default validation rules:', function () {
    beforeEach(function () {
      var fakeConfiguration = function() {
        return [

```

```

        {type:'nonPositive'},
        {type:'nonDivisible', options:{divisor: 3, error: 'error.
three'}},
        {type:'nonDivisible', options:{divisor: 5, error: 'error.
five'}}}
    ];
};

var newValidator = factoryWithConfiguration(fakeConfiguration);
validator = newValidator('default');
});

// Skipped for brevity
});
);
);

```

We just changed the setup; instead of creating the validator directly, a factory with a fake configuration is used. The fake configuration is just a plain function that returns a hard-coded rule description in JSON. This specific test double is a **stub**, because it has no logic and only returns a hard-coded response.

We need to create the `lib/factory.js` file and add the minimum code to make this new test pass:

```

var validatorWith = require('./validator'),
nonPositiveValidationRule = require('../rules/nonPositive'),
nonDivisibleValidationRule = require('../rules/nonDivisible');

module.exports = function () {
    return function () {
        return validatorWith([
            nonPositiveValidationRule,
            nonDivisibleValidationRule(3, 'error.three'),
            nonDivisibleValidationRule(5, 'error.five')
        ]);
    };
};

```

As you can notice, we have just hardcoded the original setup into the factory code. Clearly, this is not the solution; we need to add more tests to get the implementation of a real factory! For example, we should test whether we used the `'default'` rule set name to access the correct configuration:

```

describe('A validation', function () {
    var validator, configuration;
    context('using the default validation rules:', function () {
        beforeEach(function () {

```

```
configuration = function() {
  configuration.callCount++;
  configuration.args = Array.prototype.slice.call(arguments);
  return [
    {type:'nonPositive'},
    {type:'nonDivisible', options:{divisor: 3, error: 'error.
three'}},
    {type:'nonDivisible', options:{divisor: 5, error: 'error.
five'}}];
};
configuration.callCount = 0;
var newValidator = factoryWithConfiguration(configuration);
validator = newValidator('default');
});

it('will access the configuration to get the validation rules',
function() {
  expect(configuration.callCount).to.be.equal(1);
  expect(configuration.args).to.be.deep.equal(['default']);
});
// Skipped for brevity
});
});
```

First of all, `fakeConfiguration` has been renamed to `configuration` and stored in a variable. Then a new test checks whether we are calling the configuration system to get the default rule set. For this, we changed the implementation of our test double to transform it into a spy. A `spy` will record what happened to it, so we can test this information again later. Here, we are just recording how many times we have called the arguments of the last call.



There is some confusion in the terminology of test doubles. It is common to encounter the term `mock` to refer to any kind of test double. However, remember from last chapter that there are actually four kinds of test doubles: `fakes`, `stubs`, `spies`, and `mocks`.

To make the test pass:

```
module.exports = function (findConfiguration) {
  return function () {
    findConfiguration('default');
    return validatorWith([
      {type:'nonPositive'},
      {type:'nonDivisible', options:{divisor: 3, error: 'error.
three'}},
      {type:'nonDivisible', options:{divisor: 5, error: 'error.
five'}}]);
  };
};
```

```
    nonPositiveValidationRule,
    nonDivisibleValidationRule(3, 'error.three'),
    nonDivisibleValidationRule(5, 'error.five')
  );
};

};

};
```

This is better, but we are not really doing anything with the result. We need another set of tests that force us to write the logic that transforms the JSON file into actual validation rule instances. We can do this using triangulation. We will now write another scenario where we will use a different set of validation rules:

```
describe('A validation', function () {
  var validator, configuration;
  context('using the default validation rules:', function () {
    // Skipped for brevity
  });
  context('using the alternative validation rules:', function () {
    beforeEach(function () {
      configuration = function () {
        configuration.callCount++;
        configuration.args = Array.prototype.slice.call(arguments);
        return [
          {type: 'nonPositive'},
          {type: 'nonDivisible', options: {divisor: 11, error: 'error.eleven'}}
        ];
      };
      configuration.callCount = 0;
      var newValidator = factoryWithConfiguration(configuration);
      validator = newValidator('alternative');
    });
    it('will access the configuration to get the validation rules',
      function () {
        expect(configuration.callCount).to.be.equal(1);
        expect(configuration.args).to.be.deep.equal(['alternative']);
      });
    // TODO: Dear reader, add tests for this set of rules!
  });
});
```

This new scenario will force us to use the rule set name parameter and transform the JSON file into actual validation rules. Of course, we need to add the tests in the new scenario for the specified rule set. I leave the exercise of making two or three more test-first iterations and deriving a good implementation of the factory to you. After some refactoring, you will probably end up with something similar to the following lines of code:

```
var validatorWith = require('./validator'),
    nonPositiveValidationRule = require('./rules/nonPositive'),
    nonDivisibleValidationRule = require('./rules/nonDivisible');

var ruleFactoryMap = {
    nonPositive: function () {
        return nonPositiveValidationRule;
    },
    nonDivisible: function (options) {
        return nonDivisibleValidationRule(options.divisor, options.error);
    }
};

function toValidatorRule(ruleDescription) {
    return ruleFactoryMap[ruleDescription.type](ruleDescription.
options);
}

module.exports = function (findConfiguration) {
    return function (ruleSetName) {
        return validatorWith(findConfiguration(ruleSetName).
map(toValidatorRule));
    };
};
```

Of course, your implementation does not need to be exactly the same as the previous one!

Is it traditional TDD or BDD?

In small systems such as this, the distinction is very blurry, since there is a very direct correlation between features and components.

Actually, what we have done is more BDD than component unit testing. We have chosen to use test doubles to define the boundary of our feature. What we have stubbed is the configuration service of the validator; in doing so, we have chosen this interface as the boundary of our feature.

In a real system, then, we would have needed to write additional BDD specs for the configuration feature and implement it. So, we have split this hypothetical system into two features: validator and configuration. This configuration feature could be very simple, like accessing a plain JSON file; in this case, we can do a very simple integration test. Alternatively, it can be very complex, involving an interface for a business validation rule administrator and some kind of database.

If we were doing traditional TDD, we could have chosen otherwise—that is, to stub the validation rules. This way, we could have tested the validator component in isolation, using dummy rules. Then we would have tested each rule individually. In this simple example, one can argue that the individual rules do not belong to the validator feature but to the configuration feature, making BDD and TDD indistinguishable.

However, this is really not the point; traditional TDD will not help us to make decisions about which component belongs to which feature and to be explicit about how we decompose our system in features. This is not the case with BDD.

Welcome Sinon!

As you may have noticed, making test doubles involves a lot of boilerplate code and adds some complexity to our tests. But do not worry! There are a lot of libraries to create test doubles. We will use Sinon in this book. To install it, just type this:

```
$ me@~/validator> npm install --save-dev sinon
```

Alternatively, if you want to use exactly the same version as me then run the following command:

```
$ me@~/validator> npm install --save-dev sinon@1.10.2
```

However, any 1.x should work here. Now, we can replace our handmade spy with a sinon one:

```
var chai = require('chai'),
    expect = chai.expect,
    sinon = require('sinon'),
    factoryWithConfiguration = require('../lib/factory');

describe('A validation', function () {
  var validator, configuration;
  context('using the default validation rules:', function () {
    beforeEach(function () {
```

```
configuration = sinon.stub();
configuration.returns([
  {type: 'nonPositive'},
  {type: 'nonDivisible', options: {divisor: 3, error: 'error.
three'}},
  {type: 'nonDivisible', options: {divisor: 5, error: 'error.
five'}},
]);
var newValidator = factoryWithConfiguration(configuration);
validator = newValidator('default');
});

it('will access the configuration to get the validation rules',
function () {
  expect(configuration.callCount).to.be.equal(1);
  expect(configuration.calledWithExactly('default')).to.be.ok;
});
// Skipped for brevity
});

context('using the alternative validation rules:', function () {
  beforeEach(function () {
    configuration = sinon.stub();
    configuration.returns([
      {type: 'nonPositive'},
      {type: 'nonDivisible', options: {divisor: 11, error: 'error.
eleven'}}
    ]);
    var newValidator = factoryWithConfiguration(configuration);
    validator = newValidator('alternative');
  });
  it('will access the configuration to get the validation rules',
function () {
  expect(configuration.callCount).to.be.equal(1);
  expect(configuration.calledWithExactly('alternative')).to.be.ok;
});
// Skipped for brevity
});
});
```

The first thing to notice is that we are calling `sinon.stub()` to create a spy. Actually, in Sinon, when you call `sinon.stub()`, it will return an object that has the capabilities of both a spy and stub. This is what we really want, as we not only need to create a test double that records its history, but also one that returns a predefined result.

To program the test double to return a predefined result, we used the `returns(...)` method. If we had created the test double using the `sinon.spy()` method, the resulting test double would not have had the `returns(...)` method.

Sinon automatically provides a `callCount` property with the number of invocations the test double has received. It has several other properties and methods. We could have used the `calledOnce` property, such as the following:

```
expect(configuration.calledOnce).to.be.ok;
```

This tests in exactly the same way as our example but will generate a different error message if the test fails. There exist the `calledTwice` and `calledThrice` properties too.

The `calledWith()` and `calledWithExactly()` methods will return `true` if the last invocation included the specified parameters. The difference between both versions is that the latter will check whether the parameter list is exactly the one specified, and the first version will just check whether at least the specified parameters are received.



Sinon has an extensive API to create spies, stubs, and mocks. In <http://sinonjs.org/docs/>, you have an exhaustive reference for this API.



Integrating Sinon and Chai

Although `sinon` allow us to avoid the activity of writing test doubles, it has made our assertions a bit less expressive. Basically, most of our assertions on the test double will be of the `to.be.ok` type. Here, the problem is that, in the case of failure, the report message does not give a good diagnosis of what happened. Another problem is that we need to put all the expectation inside the `expect(...)` function, which is not very readable.

Fortunately, Chai is an extensible library, and there is already a bridge between Sinon and Chai; it is unsurprisingly called `sinon-chai`:

```
$ me@~/validator > npm install --save-dev sinon-chai
```

I am using version 2.5.0, but any 2.x version should be OK. Now, we will modify the test to include the test to use the new library:

```
var chai = require('chai'),
    expect = chai.expect,
    sinon = require('sinon'),
    factoryWithConfiguration = require('../lib/factory');

chai.use(require('sinon-chai'));

describe('A validation', function () {
  var validator, configuration;
  context('using the default validation rules:', function () {
    // Skipped for brevity
    it('will access the configuration to get the validation rules',
      function () {
        expect(configuration).to.have.been.calledOnce;
        expect(configuration).to.have.been.calledWithExactly('default');
      });
    // Skipped for brevity
  });

  context('using the alternative validation rules:', function () {
    // Skipped for brevity
    it('will access the configuration to get the validation rules',
      function () {
        expect(configuration).to.have.been.calledOnce;
        expect(configuration).to.have.been.calledWithExactly('alternati
ve');
      });
    // Skipped for brevity
  });
});
```

First of all, we need to install the `sinon-chai` plugin. To do so, we just need to call `chai.use(require('sinon-chai'))` before our test suite.

Then we just need to change the assertions. The idea is that you can directly pass the test double to the `expect(...)` function and then use the new assertions that `sinon-chai` provides. There will be one assertion for each method in the stub and spy interfaces of `sinon`.

If your test fails, you will see a more informative error message, explaining the difference between what we expected and what really happened. The result will look something like the one shown in the following screenshot:

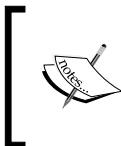
```
A validation
using the default validation rules:
1) will access the configuration to get the validation rules
   ✓ for valid numbers, will return no errors
   for not strictly positive numbers:
     ✓ like 0, will include error.nonpositive
     ✓ like -2, will include error.nonpositive
   for numbers divisible by 3:
     ✓ like 3, will include error.three
     ✓ like 15, will include error.three
   for numbers divisible by 5:
     ✓ like 5, will include error.five
     ✓ like 15, will include error.five
using the alternative validation rules:
2) will access the configuration to get the validation rules
   ✓ for valid numbers, will return no errors
   for not strictly positive numbers:
     ✓ like 0, will include error.nonpositive
     ✓ like -2, will include error.nonpositive
   for numbers divisible by 11:
     ✓ like 11, will include error.eleven
     ✓ like 22, will include error.eleven

12 passing (41ms)
2 failing

1) A validation using the default validation rules: will access the configuration to get the validation rules:
   AssertionError: expected stub to have been called with exact arguments default
   stub(defaultBUG) => [{ type: "nonPositive" }, { options: { divisor: 3, error: "error.three" } },
   type: "nonDivisible" }]
   at Context.<anonymous> (/Users/ianm/validator/test/validator-spec.js:24:42)
   at callFn (/Users/ianm/validator/node_modules/mocha/lib/runnable.js:223:21)
   at Test.Runnable.run (/Users/ianm/validator/node_modules/mocha/lib/runnable.js:216:7)
   at Runner.runTest (/Users/ianm/validator/node_modules/mocha/lib/runner.js:373:10)
   at /Users/ianm/validator/node_modules/mocha/lib/runner.js:451:13
   at next (/Users/ianm/validator/node_modules/mocha/lib/runner.js:298:14)
   at /Users/ianm/validator/node_modules/mocha/lib/runner.js:386:7
   at /Users/ianm/validator/node_modules/mocha/lib/runner.js:396:23
```

An error report with Sinon

Note that we are calling the configuration with a wrong parameter, `defaultBUG`, instead of `default`.



You can have a look at the complete documentation at <https://github.com/domenic/sinon-chai>. As you can see, for each method in a `sinon` test double, there is one equivalent method in `sinon-chai`.

Summary

Wow! This was a long chapter, but we covered a lot of ground. You learned the basics of Node and NPM, just enough to be able to set up a simple development environment for BDD. Node allows us to execute JavaScript from the command line, something that is essential to perform BDD. With NPM, we can initialize our project, manage its dependencies, and install tools and libraries, such as Mocha and Sinon.

We practiced the test-first cycle, Red/Green/Refactor, using Mocha and Chai. During your first contact with the test-first cycle, you learned not only how to use the tools and libraries but also some useful testing techniques:

- Adding duplication to your code with triangulation, to expose patterns that help you discover the correct algorithm and design you need to use.
- Knowing how a bug can lead to a new failing test or change the existing ones, to reflect our new understanding of the system.
- Organizing test code in Mocha to avoid duplication using parameterized tests and the `before` and `beforeEach` functions.
- Organizing tests using `describe`, `context`, and `it`, and how to describe them in a consistent way.
- Using test doubles to help define the boundary of our features. Contrast this with the fact that, in traditional TDD, test doubles help us define the boundary of our component.
- Making test doubles using Sinon and how to integrate them nicely with Sinon and Mocha using `sinon-chai`.

Finally, you learned that the line between component unit testing and BDD is sometimes blurry but that BDD emphasizes the decomposition of a system in features instead of components, helping locate the feature boundaries and interfaces.

In the next chapter, we will look at how to write BDD features more in depth using a more realistic example. You will also learn more advanced techniques to organize our test codebase and how to test asynchronous code.

3

Writing BDD Features

Although in the last chapter, you learned how to use Mocha, Sinon, and Chai to code some BDD tests, it was not so clear how we can write a good feature, given a set of requirements. In this chapter, we will go through the following topics:

- Exploring in greater depth how to write good features. For this, we will work on a more realistic example: myCafé, an imaginary start-up.
- Since myCafé is a JavaScript server, we will need to implement its functionality using asynchronous programming, so we will learn how to test asynchronous code.
- We will explore more techniques to organize your test codebase so that we can make it more expressive and reuse code across different features.

What we are not going to see in this chapter is the actual implementation of the system, but only the code of the tests. I expect you to write the actual code necessary to make the tests pass! This way, you will be able to practice the test-first cycle a bit more.

Introducing myCafé

The famous myCafé start-up has hired us to start developing their core business. The myCafé business idea is very simple: allow customers to preorder coffee from a mobile web page so that, when they arrive at the coffee shop, it will already be waiting.

This involves several subsystems, such as payment, orders, shop, inventory, and so on. We have been charged with the task of developing the orders subsystem.

The order subsystem is clearly not very complicated. The user goes to the ordering page, selects the products they want, and then places the order. Users can add drinks to their order, remove them, and change the quantity desired for each. The order subsystem can also place the order, triggering the payment process; after that, it will communicate the order to the shop.

We will focus only on the basic functionality mentioned earlier. We will not deal with other more advanced functionality such as having several orders or being able to select between several shops. This fits in with Agile methodologies, where you first focus on the basic features and then add extra layers of behavior in additional iterations.

So, for now, we will focus only on the ordering page, and we will assume that each user can have only one order and only one shop at a time.

Writing features

The first thing we need to do is to identify the features on the ordering page. As we saw in the last chapter, there should be only one single user action in the system per feature.

A nice trick to extract features is to identify the main conceptual entities that the user is going to interact with. Then, we can simply discover the operations that the user can execute on each entity and write one feature per operation.

In the order subsystem, we will obviously have an **Order** entity. With this in mind, we can think of several features, such as the following ones:

- Placing the order
- Creating a new order
- Adding some beverages to the order
- Removing a beverage from the order
- Changing the quantity of a beverage

Of course, each one of these actions is a different feature, but we are forgetting a very important one: displaying the order. After all, whenever the user visits the ordering page or refreshes it, they are simply performing an action against the server.

So, we actually have six features: create an order, display the order, add a beverage, remove a beverage, and change the quantity of a beverage. For brevity, we will not go into the details of all of them; instead, we will focus on displaying an order.

Displaying a customer's order

Let's start with the easiest feature: displaying an order. For this, we need to identify what information we need to display to the user. In this case, we will display the following features:

- The items in this order: the name of the beverage, the quantity, and the unit price
- The total price of the order

We will write the feature directly using Mocha. For this, we need to create a project as follows:

```
$ me@~-> mkdir mycafe && cd mycafe
$ me@~/mycafe> npm init
```

Just answer the questions to initialize `package.json` or simply edit it afterwards. Specify a test command similar to the one we saw in previous chapter. Then, install the testing libraries:

```
$ me@~/mycafe> npm install --save-dev mocha chai sinon sinon-chai
```

Finally, we will create the following folders:

```
$ me@~/mycafe> mkdir test lib
```

Inside the `test` folder, we will create a test file called `customer_displays_order.js`; this will contain our first feature:

```
'use strict';

var chai = require('chai'),
    expect = chai.expect,
    sinon = require('sinon');

describe('Customer displays order', function () {
})
```

Note that we are using the feature name for both the name of the test file itself and the title of the feature. A default format for feature names can be something like `<ROLE>_<ACTION>_<ENTITY>`, where `<ROLE>` is the name of the kind of user that performs the actions, `<ACTION>` is the user operation represented by this feature, and `<ENTITY>` is the main information entity that is affected by the feature action.

Now, we will go for our first scenario. Remember from the last chapter that one scenario represents a different setup of the system, or a different user input; this results in a different result. At the beginning, it is always interesting to start with the most simple success scenario. In the case of a display feature, this is often displaying an empty entity – in this example, an empty order. As we saw in the last chapter, to signal a scenario, we use the `context` function:

```
describe('Customer displays order', function () {
  context('Given that the order is empty', function() {
    });
});
```

Now, we need to add a test for each thing we think should happen when a user displays an empty order. It is clear that we should not see any item, and the total price should be zero:

```
describe('Customer displays order', function () {
  context('Given that the order is empty', function() {
    it('will show no order items');
    it('will show 0 as the total price');
  });
});
```

Note that we have not passed any function as a second argument of the `it` function. This is not an error, but it tells Mocha that these tests are pending. If you run Mocha now, it will not execute any tests, but it will generate a normal test report saying that the tests are pending.

It is often a good idea to make the tests pending when we are exploring the functionality of a feature. Right now, we are not interested in writing the test code; we are interested only in discovering what the behavior of the feature is so that we can obviate the actual test implementation.

Another interesting thing is the wording. Normally, the scenario title is worded in the past or in the present, expressing that the state of the system is already as described in the moment when the user performs the operation. Contrast this with tests titles that are in the future tense, emphasizing what will happen in the future when the user performs the operation. The operation itself is in the present tense, and is defined in the title of the feature; it is common to all the scenarios.

So far everything has been very obvious, but there is still one test missing. Until now, we have been writing tests that describe what information is seen by the user. However, this is not the end of the story; we need to describe which operations the user can perform on the order. Can the user add a new beverage to the order? Can the user submit the order for payment? We need to write tests that describe this. Consider that, with each change in the state of the order, some operations can be enabled, and some others can be disabled. So, we should be specific about the operation that the user can execute when an order is displayed:

```
describe('Customer displays order', function () {
  context('Given that the order is empty', function() {
    it('will show no order items');
    it('will show 0 as total price');
    it('will not be possible to place the order');
    it('will be possible to add a beverage');
    it('will not be possible to remove a beverage');
    it('will not be possible to change the quantity of a beverage');
  });
});
```

As you can see, we are explicit about the actions that are possible and the ones that are not for an empty order.

This scenario is good as it is, but it could be a bit better. It stands to reason that, if we do not have an order item, we cannot perform operations about it, such as removing the item or changing the quantity of a beverage. With this scenario, we want to express that, with empty orders, we can only add a new beverage. So, we can simplify the scenario as follows:

```
context('Given that the order is empty', function () {
  it('will show no order items');
  it('will show 0 as total price');
  it('will only be possible to add a beverage');
});
```

This is much more compact and conveys our intention better. As a rule of thumb, we should write scenarios with tests that are relevant and not redundant. The shorter the scenario, the better, since it's easier to understand and maintain, and faster to execute.



Why should we care about whether an action is available or not? The reason is simple; if we do not test this kind of thing here, where are we going to test it? Somewhere, there should be some logic to enable and disable controls, forms, and so on. Putting this logic in view (ERB, JSP, Mustache template, and so on) and deciding not to test it is not a good idea. On the other hand, UI and end-to-end tests are expensive and very slow, so we cannot use them to drive our code!

Now that the scenario is complete, we would normally start adding the test code for one test at a time and driving the implementation using the test-first cycle. When all the tests pass and the code has been cleaned up, then we start with the next scenario. Repeat this until we cannot think of any more scenarios, and we are done!

However, in this chapter, we will focus on exploring the concept of writing a feature, so we will skip the actual implementation part and continue directly with the next scenario. If we have tested for empty orders, what about the orders with actual contents? Have a look at the following code:

```
describe('Customer displays order', function () {
  context('Given that the order is empty', function() {
    // Skipped for brevity
  });

  context('Given that the order contains beverages', function() {
    it('will show one item per beverage');
    it('will show the sum of the unit prices as total price');
    it('will be possible to place the order');
    it('will be possible to add a beverage');
    it('will be possible to remove a beverage');
    it('will be possible to change the quantity of a beverage');
  });
});
```



Note that now it is still possible to add a beverage, but it is not the only thing we can do; we can add, remove, and edit the quantity of beverages in the order. We can place the order too.

Are we done with all the scenarios? Not quite yet. In most applications, the user can receive messages from the system. So, an order can have messages that need to be shown to the system. In this example, there will be error messages; once they are shown, they will not be displayed again. We can implement this with the following scenario:

```
describe('Customer displays order', function () {
  context('Given that the order is empty', function () {
    // Skipped for brevity
  });

  context('Given that the order contains beverages', function () {
    // Skipped for brevity
  });

  context('Given that the order has pending messages', function() {
    it('will show the pending messages');
    it('there will be no more pending messages');
  });
});
```

It is interesting to note that the scenario does not say anything about the actions or contents of the order. This is because these details are not relevant to the scenario. What changes in this scenario is the fact that we have pending messages. On the one hand, we do not say anything about the contents of the order, so we cannot write any test about them. On the other, we have already tested how to display orders, depending on their contents, so it would be a duplication of the tests, which will almost certainly pass anyway.

Should we write a scenario about an order with no pending messages? No, it is not necessary. On the one hand, if we implement the functionality one scenario at a time before going to the next scenario, we will find that this no messages scenario will pass directly. In the earlier chapters, we saw that we should not write tests that pass directly. On the other hand, if we write the scenario, it will be something like this:

```
context('Given that the order has pending messages', function () {
  it('will show no messages');
  it('there will be no more pending messages');
});
```

This, actually, is exactly the same test as the earlier one, since 'will show no messages' is exactly the same thing as 'will show the pending messages' when there are actually no messages to show.

The only reason we wrote different scenarios for empty and nonempty orders is because that circumstance actually changes the set of actions that the user can perform, so it actually matters.

The pending messages and order contains beverages scenarios could also benefit from a bit of triangulation. For this, we can write parameterized scenarios, as we will see at the end of this chapter.

Are we done now? No, we still need to write scenarios about operational errors. Bad things can happen, for example the database might be down or the order requested might not exist. For brevity, we will skip this kind of scenario for now.

Tips for writing features

In general, keep your features and scenarios small, concise, and relevant, and ensure that they are not redundant. This will save you a lot of time and money in maintenance and make your test suite and reports easier to read and understand.

Write your features incrementally. First, discover a new scenario, a simple one if it is possible. Then, you can focus on the tests. Add a test, make it pass, and clean your code. Add another test and so on, until you cannot think of another failing test for this scenario. Finally, try to discover another scenario where there can be failing tests. Repeat until you cannot think of another failing scenario.



What actually happens in reality is that you are going to miss some scenarios and even some tests. This is normal, since we usually do not have a complete picture of the functionality at the beginning. Do not worry about this, because we can always add a new scenario or test when we realize that something is missing. BDD is an agile and lean approach, so do not bother about making a perfect and complete feature in the first shot.

To discover which scenarios we can have, it is convenient to ask yourself whether the outcome of the user operation would be different in the following cases:

- If the system is in a different state
- If some of the systems we need to interact with do not respond in a timely fashion or simply return an error
- If some of the systems we need to interact with return different valid responses
- If the user performed a different action before the current one
- If the user has different credentials
- If the user enters incorrect information

All of these questions can lead us to discover new scenarios where we just need to change the setup and the input data. It is convenient that, in each scenario, only one thing changes at a time.

In order to understand which tests we need to add to each scenario, we need to consider:

- The new state of the system
- The actual response of the system to the user
- The set of actions that is available to the user in the new states
- Possible interactions with other systems, such as a database, a web API, and so on
- Other side-effects

Note that, for the display scenarios, there are no side-effects or interactions with other systems, so we only need to consider the actual response and the set of actions.

And the final and the most important tip is that, for each feature, there should be only one user action that will be the same across scenarios.

Starting to code the scenarios

Let's start implementing the scenarios. We can start with the simple scenario about empty orders. The first thing to do is to implement the setup of the scenario, so we need to somehow define that the order of the user is empty. To do so, we need to think a bit about the architecture of our system. It seems reasonable to have an order database of some kind and to access to it using a very thin DAO. So, we can create an order system using a test double for this DAO and then ask the order system to display a specific order:

```
var chai = require('chai'),
    expect = chai.expect,
    sinon = require('sinon'),
    orderSystemWith = require('../lib/orders');

describe('Customer displays order', function () {
  context('Given that the order is empty', function () {
    beforeEach(function () {
      var orderDAO = {};
      orderSystem = orderSystemWith(orderDAO);

      this.result = orderSystem.display('some empty order id');
    });
    it('will show no order items');
    it('will show 0 as total price');
    it('will only be possible to add a beverage');
  });
  // Skipped for brevity
});
```



Note that we also added the execution of the feature's action in the `beforeEach` function, because it is common to all the tests of the same scenario.

This seems the simplest design we can imagine. Now, we need to think a bit about the interaction between the order system and the DAO when we want to display an order. It seems reasonable that the order system will need to retrieve the order from the DAO:

```
beforeEach(function () {
  var orderDAO = {
    byId: sinon.stub()
  },
```

```

orderSystem = orderSystemWith(orderDAO);

orderDAO.byId.withArgs('some empty order id').returns([]);

this.result = orderSystem.display('some empty order id');
});

```

We added a `byId` stub method to the DAO; this method will return an empty order when asked for it. To do so, we used the `withArgs` method of Sinon's stubs; this method allows us to tell the stub about the value to return, depending on the received arguments. The chosen representation for an empty order is simply an empty array, since it is the simplest thing.

In this case, we are in the process of discovering the interface of the DAO and understanding how to represent the order at a database level. It might be possible that these things are already known. For example, we might already have a database in place, or we are using a DAO framework (ORM/ODM) that imposes some constraints on the interface of the DAO. In this case, we just need to consult the pertinent documentation and make our test double exactly like the DAO we are going to use.

Since the creation of the DAO itself and the order system is going to be common to all scenarios, we will move it to a common setup:

```

describe('Customer displays order', function () {
  beforeEach(function () {
    this.orderDAO = {
      byId: sinon.stub()
    };
    this.orderSystem = orderSystemWith(this.orderDAO);
  });
  context('Given that the order is empty', function () {
    beforeEach(function () {
      this.orderId = 'some empty order id';
      this.orderDAO.byId.withArgs(this.orderId).returns([]);
      this.result = this.orderSystem.display(this.orderId);
    });
    it('will show no order items');
    it('will show 0 as total price');
    it('will only be possible to add a beverage');
  });
});

```

Note how we used a `beforeEach` function instead of a `before` block to be sure that each scenario has a brand new test double. We do not want to mix up the setup between scenarios. It would be better to move the actual feature action to the common setup too, but we cannot do this. Each scenario can have a different setup or input data, and we cannot execute the action before the setup. So, it is better to keep it in the `beforeEach` function, right after the setup, even if this line of code is going to be duplicated across all the features.

In the preceding code, the objects are passed through the runtime context of the tests. In Mocha, the `this` keyword will point to the same object throughout all the test suites, so we can use it to store useful information. The other option is to play with the scope of local variables inside the test and context functions.

Let's write our tests now:

```
it('will show no order items', function () {
  expect(this.result).to.have.property('items')
    .that.is.empty;
});

it('will show 0 as total price', function () {
  expect(this.result).to.have.property('totalPrice')
    .that.is.equal(0);
});

it('will only be possible to add a beverage', function () {
  expect(this.result).to.have.property('actions')
    .that.is.deep.equal([
      {
        action: 'append-beverage',
        target: this.orderId,
        parameters: {
          beverageRef: null,
          quantity: 0
        }
      }
    ]);
});
```

We defined a response that will be a simple JSON object with the `items`, `totalPrice`, and `actions` fields. The `items` field is a simple array that contains a per-order entry. The `totalPrice` field should contain the sum of the prices of the items and, finally, the `actions` field is an array with the allowed actions for the order.

Each action is modeled as an object with an `action` field, which indicates the kind of action it is; a `target` field, which indicates which order the action should be applied to; and an optional `parameters` field, with the actual parameters of the action. In this case, we can pass the ID of a beverage using `beverageRef`, and how many we want using the `quantity` parameter.

Of course, there are multiple ways of modeling the response of the system, and probably you would end up with something a bit different. The important point here is that the tests force us to model the shape of the object we return as a response.

It is interesting to note that the system is not only returning data, such as the total price and items, but it also returns the allowed actions. This forces us to model the way in which we want to describe the actions.

Modeling actions is actually quite important. If we say that, in a test, there is an `append-beverage` action with `beverageRef` and `quantity`, there should exist, somewhere, a feature that tests these actions with the exact same parameters! If we change the parameters in one test, then we must change all the other features to be consistent.

Testing asynchronous features

If you have made the tests pass, you are probably thinking that you are done. But actually, you are not. The problem is that we have designed a synchronous API. This is not feasible in a JS application, because JS is single-threaded. Any I/O will cause our system to block, and we need to process the requests one at a time. This is not acceptable in a server or in a UI application. So, we need to change our design to use an asynchronous API, but how do we test an asynchronous API?

Testing a callback-based API

Instead of returning the value directly, we can change our API to use callbacks. The same thing applies to our DAO. After all, the DAO will perform IO, so it needs to be asynchronous. So, let's change our test. For this, we first need to change the setup and action:

```
context('Given that the order is empty', function () {
  var result;
  beforeEach(function (done) {
    this.orderId = 'some empty order id';
    this.orderDAO.byId
```

```
        .withArgs(this.orderId)
        .callsArgWithAsync(1, null, []);

    this.orderSystem.display(this.orderId, function (err, res) {
      result = res;
      done(err);
    });
  });
it('will show no order items', function () {
  expect(result).to.have.property('items').that.is.empty();
});
// Skipped for brevity
});
```

The first important thing to note is that now the function that we pass in `beforeEach` receives a `done` parameter. This parameter is a callback that we must call when our code has finished. The functions that we pass to `it`, `beforeEach`, `afterEach`, `before`, and `after`, can receive this extra parameter. So, we can use asynchronous code not only in the tests, but in the setup and cleanup functions too.

Mocha will wait for the `done` callback to be invoked before proceeding with the test suite. If we call it without parameters, with `null` or `undefined`, Mocha will mark the test as passing. If we supply an error when calling the callback, Mocha will mark the test as failed and report the error. Another way in which we can fail is due to a timeout. If we take too long to call the `done` callback, Mocha will timeout and report an error.



We can use the `--timeout` or `-t` parameter in the Mocha command line to define how long this timeout will be. This will set up a global timeout value for all of our tests. We can also specify the timeout at a suite or at a test level. To do so, we can use the `this.timeout(milliseconds)` method of Mocha. We can call it inside a `describe` block to set the timeout at the test-suite level. We can also call it inside an `it` block to set the timeout only for that test.

The `display` method of the order system has been changed to use a callback to return the result. The Node.js convention for callbacks is used. In this convention, the error is always the first parameter, and the result is the second one. If there is no error, this parameter would simply be `null`, so we can directly call the `done` callback using the `error` parameter.

A bit of the code has been changed to use the `result` local variable to hold the result, instead of the runtime context of the test, since it is more convenient when working with callbacks. Obviously, the tests have been changed to refer to `result` instead of `this.result`.

The other important change is in the way we set up the DAO. First, we did not specify the callback in the `withArgs` function. This is not a problem, since `withArgs` is not a strict matcher and will not check whether the number of arguments actually passed to the stub are exactly the same as specified.

Then, we used `callsArgWithAsync` to tell the stub to call the callback it receives as the second parameter. We need to specify the index of the parameter that will be the callback and then the actual parameters that will be passed to the callback. Since we are using the Node.js convention, the parameters are `null`, which means that there is no error, and the empty array, which means it is an empty object.

When we use `callsArgWithAsync`, the stub will call the specified callback when it is invoked, but not immediately. It will schedule the callback invocation for the next tick. This means that the JS runtime will wait for the current execution to finish before making the invocation.

There is another version of this function; it is called `callsArgWith` and will call the callback immediately the moment the stub is invoked. However, it is better to use the asynchronous version, since it is more realistic. A real DAO will not invoke the callback immediately, but it will wait until it has retrieved the information from the database.



Both versions, `callsArgWith` and `callsArgWithAsync`, will fail if the parameter supplied in the specified position is not a function.

Testing a promise-based API

There is another way of modeling an asynchronous API: using promises. Callbacks are difficult to compose unless you are comfortable with functional programming. The naïve way of composing asynchronous functions that use callbacks is to nest one callback inside another. This can very easily lead to callback hell: spaghetti code, memory leaks, and a control flow that is difficult to follow.

You can actually combine two callback-based functions safely using functional programming, but there are other options. The most popular one is to use promises.

If you do not know anything about promises, just keep reading. If you already know about them, just skip the following section.

Interlude – promises 101

This section will give you a very short introduction to promises. A **promise** is an object that represents the eventual result of an asynchronous function. Let's look at an example:

```
var promisedUser = userDAO.byId(userId);
promisedUser.then(function(user) {
  console.log(user.name);
});
```

When we call the `byId` method of the user DAO, an asynchronous process will be started in the background; this process will perform the IO necessary to retrieve the user data from the database. Since this IO will take a while and JS is single-threaded, we cannot block and wait for the IO to finish. Instead, a promise object is returned immediately, representing the eventual result of the database access.

The client code stores the promise in the `promisedUser` variable and attaches a callback to the promise using a method called `then`. All the `Promise` objects have a `then` method that we can use to register a callback; this method will be invoked only once, whenever the result is ready.

Promises can be in three states: fulfilled, rejected, and pending. They are explained here:

- A promise is in the **pending** state if the process represented by the promise is still executing, and it has not yet finished
- A promise passes from the pending state to the **fulfilled** state when the asynchronous process finishes successfully, and its result is ready
- A promise passes from the pending state to the **rejected** state when the asynchronous process finishes with an error

The promise contract assures us that the callbacks registered with the `then` method will be invoked only once when the promise is in the fulfilled state. It does not matter whether the promise was in the pending or fulfilled state when we registered the callback; we can trust that our callback will receive the result only once. This way, our code does not need to concern itself with whether the promise is already fulfilled or not; we are not going to lose the result. Compare this with Node.js streams or normal events where we can lose data if we register our callback too late.

Another interesting property of promises is that we can return a value inside the callback; here is an example:

```
var promisedUserName = userDAO
    // Returns a promise of the user
    .byId(userId)
    // Returns a promise of the user's name
    .then(function(user) {
        // The user's name value will be wrapped as a promise
        return user.name;
    });
promisedUserName.then(function(userName) {
    console.log(userName);
});
```

What happens is that the `then` method always returns a new promise. This new promise will be fulfilled when the resulting value is returned by the callback. If the callback throws an error, this second promise will be rejected.

However, what it is really more interesting is that the callback itself can return a new promise instead of an immediate value. When a callback returns a promise, the `then` method will return this promise directly. This can be very useful for composing asynchronous processes:

```
var promiseAvatarIsRendered = userDAO
    .byId(userId)
    // Returns a promise of the user's picture
    .then(function(user) {
        // pictureDAO.byId returns a promise itself
        // No extra wrapping will happen
        return pictureDAO.byId(user.pictureId);
    })
    .then(function(userPic) {
        return ui.renderUserAvatar(userPic);
    });
});
```

First, we get a promise for the user. Then, when it is fulfilled, we return a promise for the picture of the user; when we have the picture loaded, we render it. The final result is a promise that will be fulfilled whenever the whole pipeline, including rendering the avatar, is finished. Compare this code with a naïve version using callbacks:

```
function renderUserAvatar(userId, cb) {
    userDAO.byId(userId, function(err, user) {
```

```
pictureDAO.byId(userId, function(err, userPic) {
    ui.renderUserAvatar(userPic, cb);
});
});
}
}
```

How do we handle errors? We can register a second callback in the error handler; this callback will be invoked if the promise is rejected. Again, it will be invoked only once, and we will not lose the error. Here is an example:

```
var promiseAvatarIsRendered = userDAO
    .byId(userId)
    .then(function(user) {
        return pictureDAO.byId(user.pictureId);
    }, function(err) {
        if(shouldRecoverError(err))
            return defaultPicture;
        throw err;
})
    .then(function(userPic) {
        return ui.renderUserAvatar(userPic);
})
;
```

The error handlers are very simple. If we can recover the error, we should return a value. In this case, the returned promise will not be rejected, but will be fulfilled with the new value. If we cannot handle the error, we can throw it or throw a new one. In this case, the promise will be rejected with the error we throw.

Obviously, promises are a much better choice for our API, since we can both compose asynchronous processes and handle errors much easily.

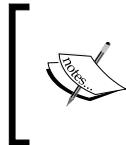
Promises are not implemented in any of the standard libraries that come with the current version of the JavaScript language (ES5). They will be in the next version, ES6. Node.js v0.10.x does not include promises either. However, do not worry; there are a lot of libraries that implement the promise that we can use. Some popular libraries for promises are Q, bluebird, and when, although there are more! All of them are good, but in this book we will use Q. Since we want to use promises for our API, we will install it:

```
$ me@~/mycafe> npm install --save q
```

Since we want to use Q for the API of our order system, we need to save it as a runtime dependency instead of as a development dependency. The preceding command will simply install the most recent stable version. If you wish to install the exact version of Q that I am using, then change the command as follows:

```
$ me@~/mycafe> npm install --save q@1.0.1
```

Anyway, you can use another promise's library if you wish!



If you want to go deep into promises, you could go to <http://promisesaplus.com/> and read the specification; alternatively, and maybe better, you just can read the documentation for the Q framework at <http://documentup.com/kriskowal/q/>.



Mocha and promises

You can test promises in Mocha very easily. This time, you do not need to invoke a done callback; you just need to return the promise! Mocha will wait for the promise to be fulfilled or rejected before proceeding with the test suite. If the promise is rejected, it will report that the test has failed using the error that rejects the promise.

It should be easy then to change the test:

```
var result;
beforeEach(function () {
  this.orderId = 'some empty order id';
  this.orderDAO.byId
    .withArgs(this.orderId)
    .callsArgWithAsync(1, null, []);

  return this.orderSystem.display(this.orderId)
    .then(function (res) {
      result = res;
    });
});
```

The change is very simple; the done callback has been removed; now, the `display` method returns a promise, so we do not need to pass a callback parameter. We used the returned promise to capture the final result into the `result` variable. Note that this callback has no parameter for errors, because the `then` method will only call it when the operation is successful. Finally, we returned the promise, so Mocha will wait for it.

Another way of testing it is to wait for the promise in the test methods. The first thing to do is to store the promise result of the `display(this.orderId)` action into a field of the runtime context called `this.result`:

```
context('Given that the order is empty', function () {
  var orderId;
  beforeEach(function () {
    orderId = 'some empty order id';
    this.orderDAO.byId
      .withArgs(orderId)
      .callsArgWithAsync(1, null, []);
    this.result = this.orderSystem.display(orderId);
  });
  // Skipped for brevity
});
```

Note that we also stored the identifier of the order for future reference in the `orderId` variable. Now, we can return a promise with an assertion in each test of this scenario. For example, to test that there are no items in the result, we can use the following lines of code:

```
it('will show no order items', function () {
  return this.result.then(function (result) {
    expect(result).to.have.property('items')
      .that.is.empty;
  });
});
```

In this solution, we just wrapped the assertions of each test in the `then` method and returned the resulting promise. Again, Mocha will wait for the promise to finish. If the assertion fails, it will throw an error; this will reject the promise, and Mocha will mark the promise as failed. We can apply the same technique to the other two tests:

```
it('will show 0 as total price', function () {
  return this.result.then(function (result) {
    expect(result).to.have.property('totalPrice')
      .that.is.equal(0);
  });
});
```

```
it('will only be possible to add a beverage', function () {
  return this.result.then(function (result) {
    expect(result).to.have.property('actions')
      .that.is.deep.equal([
        {
          action: 'append-beverage',
          target: orderId,
          parameters: {
            beverageRef: null,
            quantity: 0
          }
        }
      ]);
  });
});
```

However, this solution is not so good, since it has more boilerplate code in the tests than the other approach. On the other hand, the `beforeEach` block looks much better. Can we have the best of both approaches? Yes, we can.

Chai-as-Promised

The `chai-as-promised` package is an extension of Chai. It adds to Chai the capability to handle promises directly. It basically adds the `eventually` chain, the `fulfilled`, `rejected` properties, and the `rejectedWith()` assertion.

We can insert the `eventually` chain in our Chai assertion; from this point, `chai-as-promised` will simply wrap the rest of the assertion in the `then` method of the promise. For example, have a look at the following assertion:

```
return this.result.then(function (result) {
  expect(result).to.have.property('totalPrice')
    .that.is.empty;
});
```

The preceding assertion can be simplified as follows:

```
return expect(this.result).to.eventually
  .have.property('totalPrice').that.is.equal(0);
```

On the other hand, the `fulfilled` property will simply check whether the promise has been already fulfilled:

```
return expect(this.result).to.be.fulfilled;
```

In the same line, the `rejected` property will check whether the promise has been rejected, and `rejectedWith()` will additionally check the error, just like a normal `throw` assertion. For example, take a look at the following lines of code:

```
return expect(this.result).to.be.rejected;
return expect(this.result).to.be.rejectedWith(NotFoundError);
return expect(this.result).to.be.rejectedWith(NotFoundError,
    orderId);
```

It's now time to code. First of all, let's install `chai-as-promised`:

```
$ me@~/mycafe> npm install --save-dev chai-as-promised
```

Then, we need to import it in our test and plug it into Chai:

```
var chai = require('chai'),
    expect = chai.expect,
    sinon = require('sinon'),
    orderSystemWith = require('../lib/orders');

chai.use(require("chai-as-promised"));
// Skipped for brevity
```

Now, we can change our tests:

```
context('Given that the order is empty', function () {
    // Skipped for brevity
    it('will show no order items', function () {
        return expect(this.result).to.eventually
            .have.property('items').that.is.empty;
    });
    it('will show 0 as total price', function () {
        return expect(this.result).to.eventually
            .have.property('totalPrice').that.is.equal(0);
    });
    it('will only be possible to add a beverage', function () {
        return expect(this.result).to.eventually
            .have.property('actions')
            .that.is.deep.equal([
                {
                    action: 'append-beverage',
                    target: orderId,
                    parameters: {
```

```

        beverageRef: null,
        quantity: 0
    }
}
);
});
);

```

As you can see, the assertions now look almost like normal asynchronous assertions. You just need to add the `eventually` chain and return the resulting assertion. It is very important that you return the assertion; if not, Mocha will think that your test is synchronous and will not wait for it. This is bad because the test will always pass, since the actual assertion is performed inside a promise.



I recommend that you have a look at the GitHub project page for a complete reference of this library at <https://github.com/domenic/chai-as-promised/>.



Test doubles with promises

Sometimes, you need to make a test double for an object that returns promises instead of using callbacks. We can explore this if we assume that our DAO uses promises.

The easiest way is to make the Sinon stub return a promise if we are using the Q promises package, as shown in the following code:

```

beforeEach(function () {
  orderId = 'some empty order id';
  this.orderDAO.byId
    .withArgs(orderId)
    .returns(Q.fulfill([]));

  this.result = this.orderSystem.display(orderId);
});

```

In this case, we are using `Q.fulfill(val)` to create a promise for a normal value, which is an empty array in this case. We could have used `Q.reject(err)` if we had wanted the stub to simulate an error.

This approach is very simple and works in most cases. However, it has a problem; it is synchronous! Both methods will fulfill or reject the promise immediately. So, when we call the stub, it returns a promise that is already fulfilled.

If we want a more realistic test double, we need to ensure that the promise returned is not immediately done. This is exactly the same thing we had with `callsArgWith` and `callsArgWithAsync`. We can get something similar with a couple of utilities:

```
function promiseFor(value) {
  return Q.delay(1).then(function () {
    return value;
 ));
}

function failingPromiseWith(error) {
  return Q.delay(1).then(function () {
    throw error;
 ));
}
```

These utilities simply wrap a value or an error in a promise, but with a small delay of a millisecond. This way, it will be fulfilled or rejected asynchronously. Now, our test would be like this:

```
beforeEach(function () {
  orderId = 'some empty order id';
  this.orderDAO.byId
    .withArgs(orderId)
    .returns(promiseFor([]));

  this.result = this.orderSystem.display(orderId);
});
```

Now, our test double returns a really asynchronous promise.



Should we use a DAO based on callbacks or promises? As we saw, if we have already selected the technology that we are going to use as the database, we should make the DAO mimic the real object offered by the database driver. Usually, they offer a callback-based interface; in this case, we will stick with the callback. If you are not sure about it, you should stick to a callback-based interface because it is the most common.

Organizing our test code

We will continue coding our tests, assuming that our API uses promises but that the DAO uses callbacks. If you tried to make your first scenario pass, then you would end up with something similar to what I have in my project in the `lib/orders.js` file:

```
var Q = require('q');

module.exports = function () {
  return {
    display: function (orderId) {
      return Q.fulfill({
        items: [],
        totalPrice: 0,
        actions: [
          {
            action: 'append-beverage',
            target: orderId,
            parameters: {
              beverageRef: null,
              quantity: 0
            }
          }
        ]
      });
    }
  };
};
```

No `orderDAO` anywhere! In fact, we can remove all references to `orderDAO` in the setup or our test! What happened is that we introduced it prematurely. This is something we would not have done in real circumstances but, for the purposes of this book, it was very convenient.



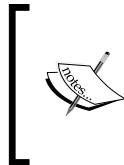
On the other hand, start thinking of the high-level architecture of our system; it is not a bad thing, provided that you timebox it. It stands to reason that the orders must have come from some kind of database. So, introducing a DAO for this, even a bit prematurely, helps us shape the boundary of our system, so it is not so bad!

We can continue writing our feature. We can write the setup and action for the next scenario:

```
context('Given that the order contains beverages', function () {
  beforeEach(function () {
    this.orderId = 'some non empty order id';
    this.orderDAO.byId
      .withArgs(this.orderId)
      .callsArgWithAsync(1, null, [
        {
          beverage: {
            id: "expresso id",
            name: "Expresso",
            price: 1.50
          },
          quantity: 1
        },
        {
          beverage: {
            id: "mocaccino id",
            name: "Mocaccino",
            price: 2.30
          },
          quantity: 2
        }
      ]);
    this.result = this.orderSystem.display(this.orderId);
  });
  // Skipped for brevity
});
```

This time, the DAO will return an order with two items. The first item is a single espresso, and the second one is two mocaccinos. Each item is composed of a quantity and a beverage. The beverage has an identifier, a name, and a price.

Again, the simple fact that we need to make a setup forces us to think about the design of our system. In this case, we need to figure out how the order items are stored in the database.



If the data schema for the order is already known, just be faithful to it. Remember that DAOs should not involve any logic beyond the mere IO to retrieve or update the information. Any data transformation or additional logic should be inside our order subsystem. This way, we can cover such logic with the tests we are writing.

In this example, we have chosen a specific data schema where the beverage information is embedded in the order document. We could have chosen another approach such as using a reference to the beverage instead of a copy. In a real project, this is a good moment to have a technical discussion about which data schema design to use.

Irrespective of the data schema we use for our storage, the result of the display should include all the relevant data that we want to display to the user. So, the tests for the order information are as follows:

```
it('will show one item per beverage', function () {
  return expect(this.result).to.eventually
    .have.property('items').that.is.deep.equal([
      {
        beverage: {
          id: "expresso id",
          name: "Expresso",
          price: 1.50
        },
        quantity: 1
      },
      {
        beverage: {
          id: "mocaccino id",
          name: "Mocaccino",
          price: 2.30
        },
        quantity: 2
      }
    ]);
});

it('will show the sum of the unit prices as total price', function () {
  return expect(this.result).to.eventually
    .have.property('totalPrice').that.is.equal(6.10);
});
```

The actual test for the order contents and the setup has a lot of duplication, so it is better to clean it a bit:

```
beforeEach(function () {
    this.orderId = 'some non empty order id';
    this.expresso = {
        id: "expresso id",
        name: "Expresso",
        price: 1.50
    };
    this.mocaccino = {
        id: "mocaccino id",
        name: "Mocaccino",
        price: 2.30
    };
    this.orderItems = [
        { beverage: this.expresso, quantity: 1},
        { beverage: this.mocaccino, quantity: 2}
    ];
    this.orderDAO.byId
        .withArgs(this.orderId)
        .callsArgWithAsync(1, null, this.orderItems);

    this.result = this.orderSystem.display(this.orderId);
});

it('will show one item per beverage', function () {
    return expect(this.result).to.eventually
        .have.property('items')
        .that.is.deep.equal(this.orderItems);
});
```

We simply stored the setup data in the runtime context so that we can reference it from both the tests and the setup.

The storage object pattern

Although we do not have much duplication now, we have ended up with a quite verbose setup that it is not very nice to read. Furthermore, if we want to make a similar setup in another scenario or in another feature, we will end up copying and pasting this code. This is highly probable, since most of our features will involve some kind of access to the orders database. This approach to setup is clearly not very maintainable.

What we need is to encapsulate the setup logic so that we can reuse it across all of our tests. We can start with an extract method:

```
function orderAlreadyContainsItems(orderDAO, orderId, items) {
    orderDAO.byId
        .withArgs(orderId)
        .callsArgWithAsync(1, null, items);
    return items;
}

context('Given that the order contains beverages', function () {
    beforeEach(function () {
        this.orderId = 'some non empty order id';
        this.espresso = {
            id: "expresso id",
            name: "Expresso",
            price: 1.50
        };
        this.mocaccino = {
            id: "mocaccino id",
            name: "Mocaccino",
            price: 2.30
        };
        this.orderItems = orderAlreadyContainsItems(this.orderDAO, this.
orderId, [
            { beverage: this.espresso, quantity: 1},
            { beverage: this.mocaccino, quantity: 2}
        ]);

        this.result = this.orderSystem.display(this.orderId);
    });
    // Skipped for brevity
});

```

This would be OK if we only had one method for our DAO but, as a general rule, we can use the DAO to delete, update, or create. It is not something that we are going to do in the display feature, but we will definitely need it in place to order a beverage, add a beverage, and so on.

It is common to have more than one DAO. So, we would like to have something that we can reuse for other DAOs.

For the sake of brevity, we will look at how to make a better setup now and leave implementing all of these features till later.

The idea is that we can create a test double that represents not the DAO but the external storage system, such as a database or a remote third-party service. We can create a utility in the `test/support/storageDouble.js` file with the following code:

```
'use strict';

var sinon = require('sinon');

module.exports = function () {
  var dao = { byId: sinon.stub() },
    storage = {};

  storage.dao = function () {
    return dao;
  };

  storage.alreadyContains = function (entity) {
    var data = entity.data;
    dao.byId
      .withArgs(entity.id)
      .callsArgWithAsync(1, null, data);
    return entity;
  };

  return storage;
};
```

In this file, we are creating an object that encapsulates both the setup code and the test double creation.

We created a double for the order DAO and stored it in a `dao` local variable. We offered a read accessor for this DAO too. This is good because, if we decide that we do not want to use Chai anymore or we change the interface of the DAO, we do not need to review all of our tests to accommodate the change; we can review only this file.

Then we come to the `alreadyContains` method. This method contains the setup logic to specify that there is already an order in the database. This way, if the interface of the DAO changes, we do not need to change our tests; we only need to change the `storageDouble` library. If, in the future, we need to add a new kind of setup, we can add a new method here. Additionally, we can reuse the `storageDouble` library to create a test double for any DAO that has the same interface as our orders DAO.

Let's change our test to leverage this:

```
var chai = require('chai'),
    expect = chai.expect,
    newStorage = require('../support/storageDouble'),
    orderSystemWith = require('../lib/orders');

chai.use(require("chai-as-promised"));

describe('Customer displays order', function () {
  beforeEach(function () {
    this.orderStorage = newStorage();
    this.orderSystem = orderSystemWith(this.orderStorage.dao());
  });
})
```

Note that we no longer import the Sinon module; instead, we import our storageDouble utility. We use it to create a new order storage double, and then we use the dao accessor to initialize our order system. The rest of the test ends up being like this:

```
context('Given that the order is empty', function () {
  beforeEach(function () {
    this.order = this.orderStorage.alreadyContains({
      id: 'some empty order id',
      data: []
  });

    this.result = this.orderSystem.display(this.order.id);
  });

  // Skipped for brevity

  it('will only be possible to add a beverage', function () {
    return expect(this.result).to.eventually
      .have.property('actions')
      .that.is.deep.equal([
        {
          action: 'append-beverage',
          target: this.order.id,
          parameters: {
            beverageRef: null,
            quantity: 0
          }
        }
      ])
  })
})
```

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

context('Given that the order contains beverages', function () {

    beforeEach(function () {
// Skipped for brevity
        this.order = this.orderStorage.alreadyContains({
            id: 'some non empty order id',
            data: [
                { beverage: this.espresso, quantity: 1},
                { beverage: this.mocaccino, quantity: 2}
            ]
        });

        this.result = this.orderSystem.display(this.order.id);
    });

    it('will show one item per beverage', function () {
        return expect(this.result).to.eventually
            .have.property('items')
            .that.is.deep.equal(this.order.data);
    });
    // Skipped for brevity
});
// Skipped for brevity
});
```

Note that the test is now more concise and legible. On the one hand, the order data has been consolidated into the `this.order` field. On the other, we are using the `alreadyContains` method of `orderStorage`; this makes the setup more expressive.

The example factory pattern

Let's think about our test data. It will be much better if we have a common set of standard test-beverage samples. This way, we can reference this beverage sample from any of our tests, making them less verbose.

To do so, we can create a factory object that we can use to create examples of test data. This example factory should offer factory methods that allow us, in a simplified but expressive way, to describe the data we want to have in the examples. This also applies to examples for the input data or the expected results.

We can apply this idea to our tests. First, we can create a `support/examples/beverages.js` file:

```
'use strict';

module.exports = {
  espresso: function () {
    return {
      id: "espresso id",
      name: "Expresso",
      price: 1.50
    };
  },
  mocaccino: function () {
    return {
      id: "mocaccino id",
      name: "Mocaccino",
      price: 2.30
    };
  }
};
```

This code example uses functions in order to return a new copy of the data; this way, we can change it without problems in the tests if we need to do so. Instead of using one single method capable of creating any beverage, we have opted to model only a limited set of beverages. This is much more simple and expressive. The trick is to use realistic examples taken from the problem domain.

Using example factories gives us another advantage: it protects our test from data schema changes. We can reuse this dictionary of beverages across tests; if the data schema for beverages changes, there is only one place we need to change.

💡

Do not overdo it! The point of this technique is to simplify your test codebase and make it more maintainable and expressive. Do not end up adding complex logic here or creating ultraflexible parametrizable factories. So, proceed with common sense here and define only simple and expressive factories, with only a bit of logic or none at all.

We can try to do the same with orders. In this case, we would like to create empty and nonempty orders. We can define another utility in `support/examples/orders.js`:

```
'use strict';

var beverage = require('./beverages');

var counter = 0;

function asOrderItem(itemExample) {
  return {
    beverage: beverage[itemExample.beverage](),
    quantity: itemExample.quantity
  };
}

module.exports = {
  empty: function () {
    return {
      id: "<empty order>",
      data: []
    };
  },
  withItems: function (itemExamples) {
    counter += 1;
    return {
      id: "<non empty order " + counter + ">",
      data: itemExamples.map(asOrderItem)
    };
  }
};
```

This utility is a bit more complex. It has two methods: one to generate empty orders and the other, `withItems`, to generate nonempty ones from an array of item examples.

Each of these examples consists of a beverage name and a quantity. We will transform it into a real array of items using the `asOrderItem` function that will ask the beverage examples to create the correct beverage for each name. This design helps us to abstract the test from the exact way we are managing the relationship between order items and beverages from the tests. If we change from using a embedded beverages to using references, we only need to change this utility.

The counter variable is used inside the `withItems` method to generate the id of the order, as this detail is not important in the tests.

Now, we need to change the test to leverage these utilities:

```
var chai = require('chai'),
    expect = chai.expect,
    newStorage = require('./support/storageDouble'),
    order = require('./support/examples/orders'),
    orderSystemWith = require('../lib/orders');

chai.use(require("chai-as-promised"));

describe('Customer displays order', function () {
    // Skipped for brevity
    context('Given that the order is empty', function () {
        beforeEach(function () {
            this.order = this.orderStorage
                .alreadyContains(order.empty());
            this.result = this.orderSystem.display(this.order.id);
        });
        // Skipped for brevity
    });

    context('Given that the order contains beverages', function () {
        beforeEach(function () {
            this.order = this.orderStorage
                .alreadyContains(order.withItems([
                    { beverage: 'expresso', quantity: 1},
                    { beverage: 'mocaccino', quantity: 2}
                ]));
            this.result = this.orderSystem.display(this.order.id);
        });
        // Skipped for brevity
    });
    // Skipped for brevity
});
```

As you can see, we used the order example factory in the setup. We do not need to worry about identifiers, and we can reference the beverage now. The setup is now much more compact, readable, and relevant.

You should hide any information and data structures that are not strictly relevant to the behavior of the feature we are trying to test. This includes technical details such as identifiers, for example. However, the border between relevant and irrelevant can sometimes be quite blurry.

In this specific example, is the price of each beverage relevant? Yes, you need to know the price to understand whether the total price is correctly calculated or not. From this point of view, we should have specified the price of the beverage in the setup of the tests. In this example, I decided to be less verbose at the cost of being less explicit. You can do this only if there is a strong common understanding about the test data; thus, if it is clear that a Mocaccino example always costs 230, then we do not need to be explicit about the price. This usually happens when the data is realistic and comes from the problem domain. If your team does not have this common understanding about the test examples, I recommend that you go for a more verbose and more explicit approach, hiding only the things that are not strictly relevant to the test.

Finishing the scenario

Now we can try to finish our scenario. The tests on the available actions are pending, so it is time to implement them. We can first do the test relative to being able to place the order:

```
it('will be possible to place the order', function () {
  return expect(this.result).to.eventually
    .have.property('actions')
    .that.include({
      action: 'place-order',
      target: this.order.id
    });
});
```

We simply checked whether the actions property is a collection that includes the relevant action. We can now add tests to add and remove a beverage from the order:

```
it('will be possible to add a beverage', function () {
  return expect(this.result).to.eventually
    .have.property('actions')
    .that.include({
```

```

        action: 'append-beverage',
        target: this.order.id,
        parameters: {
            beverageRef: null,
            quantity: 0
        }
    });
});

it('will be possible to remove a beverage', function () {
    return expect(this.result).to.eventually
        .have.property('actions')
        .that.include({
            action: 'remove-beverage',
            target: this.order.id,
            parameters: {
                beverageRef: beverage.espresso().id
            }
        })
        .and.that.include({
            action: 'remove-beverage',
            target: this.order.id,
            parameters: {
                beverageRef: beverage.mocaccino().id
            }
        });
});

```

These tests have exactly the same structure as the one relating to placing the order. The only interesting thing here is the remove-beverage test. Here, we are checking whether we have two remove-beverage actions, one for each beverage. We will use the and chain from Chai to make the assertion more compact. Finally, we will add a test to check whether we have an action to edit the quantity of each beverage:

```

it('will be possible to change the quantity of a beverage', function
() {
    return expect(this.result).to.eventually
        .have.property('actions')
        .that.include({
            action: 'edit-beverage',
            target: this.order.id,
            parameters: {

```

```
        beverageRef: beverage.espresso().id,
        newQuantity: 1
    }
})
.and.that.include({
    action: 'edit-beverage',
    target: this.order.id,
    parameters: {
        beverageRef: beverage.mocaccino().id,
        newQuantity: 2
    }
});
});
```

To build the actions on adding and removing beverages, we need the identifiers of the corresponding beverages, and the beverage example factory is used to get them. The tests are readable but a bit verbose. There is also a bit of duplication between all the actions, since all have a target set to the order.

We can make it a bit more succinct if we create a new example factory for actions. If we look carefully, all the actions are very much coupled to the actual contents of the order. It makes sense to create action examples from an order example. We can edit `test/support/examples/orders.js` to include a method to create order actions:

```
module.exports = {
    empty: function () {
        // Skipped
    },
    withItems: function (itemExamples) {
        // Skipped
    },
    actionsFor: function (order) {
        return {
            removeItem: function (index) {
                var item = order.data[index];
                return {
                    action: 'remove-beverage',
                    target: order.id,
                    parameters: {
                        beverageRef: item.beverage.id
                    }
                };
            }
        };
    }
},
```

```

editItemQuantity: function (index) {
  var item = order.data[index];
  return {
    action: 'edit-beverage',
    target: order.id,
    parameters: {
      beverageRef: item.beverage.id,
      newQuantity: item.quantity
    }
  };
},
appendItem: function () {
  return {
    action: 'append-beverage',
    target: order.id,
    parameters: {
      beverageRef: null,
      quantity: 0
    }
  };
},
place: function () {
  return {
    action: 'place-order',
    target: order.id
  };
}
};
}
;

```

Now, we can take an order example and, using `actionsFor`, we can create an example factory for this particular order.

The `place` and `appendItem` methods will create an action for "placing an order" and "adding a beverage" respectively. The example factory knows about the order identifier, so it can fill the `target` field.

The same thing happens with `removeItem` and `editItemQuantity`. Instead of passing the expected default value for the `newQuantity` field and the beverage identifier for the `beverageRef` field, we chose to simply pass the index of the item. This makes sense, since the item has all the information needed to fill the parameters of the action.

We can edit our scenario as follows:

```
context('Given that the order contains beverages', function () {
  beforeEach(function () {
    this.order = this.orderStorage
      .alreadyContains(order.withItems([
        { beverage: 'expresso', quantity: 1},
        { beverage: 'mocaccino', quantity: 2}
      ]));
    this.orderActions = order.actionsFor(this.order);
    this.result = this.orderSystem.display(this.order.id);
  });

  // Skipped for brevity

  it('will be possible to place the order', function () {
    return expect(this.result).to.eventually
      .have.property('actions')
      .that.include(this.orderActions.place());
  });

  it('will be possible to add a beverage', function () {
    return expect(this.result).to.eventually
      .have.property('actions')
      .that.include(this.orderActions.appendItem());
  });

  it('will be possible to remove a beverage', function () {
    return expect(this.result).to.eventually
      .have.property('actions')
      .that.include(this.orderActions.removeItem(0))
      .and.that.include(this.orderActions.removeItem(1));
  });

  it('will be possible to change the quantity of a beverage', function () {
    return expect(this.result).to.eventually
      .have.property('actions')
      .that.include(this.orderActions.editItemQuantity(0))
      .and.that.include(this.orderActions.editItemQuantity(1));
  });
});
```

Now, our scenario is much less verbose.

Parameterized scenarios

Sometimes, you would like to execute the same scenario with different sets of setup data and/or with different inputs. Maybe you need to triangulate to drive a more realistic implementation, or you would simply like to explicitly specify what happens with some edge cases.

We can try to use the technique we saw in the last chapter with our "nonempty order" scenario:

```

function scenarioOrderContainsBeverages(testExample) {
    context('Given that the order contains ' + testExample.title,
    function () {
        beforeEach(function () {
            this.order = this.orderStorage.alreadyContains(order.
                withItems(testExample.items));
            this.orderActions = order.actionsFor(this.order);

            this.result = this.orderSystem.display(this.order.id);
        });

        it('will show one item per beverage', function () {
            // Skipped, no changes here
        });

        it('will show the sum of the unit prices as total price', function
        () {
            return expect(this.result).to
                .eventually.have.property('totalPrice')
                .that.is.equal(testExample.expectedTotalPrice);
        });

        // Skipped (No changes in the rest of the tests)
    });
}

[
{
    title: '1 Expresso and 2 Mocaccino',
    items: [
        { beverage: 'expresso', quantity: 1},
        { beverage: 'mocaccino', quantity: 2}
    ],
    expectedTotalPrice: 6.10
}
].forEach(scenarioOrderContainsBeverages);

```

Now that our test is parameterized, we can add another example. However, first, let's add another beverage in the support/examples/beverages.js file:

```
module.exports = {
  // Skipped for brevity
  capuccino: function () {
    return {
      id: "capuccino id",
      name: "Capuccino",
      price: 2
    };
  }
};
```

Now, we can create a new example that contains a capuccino:

```
[  
  {  
    title: '1 Expresso and 2 Mocaccino',  
    items: [  
      { beverage: 'expresso', quantity: 1},  
      { beverage: 'mocaccino', quantity: 2}  
    ],  
    expectedTotalPrice: 6.10  
  },  
  {  
    title: '1 Mocaccino, 2 expressos, and 1 capuccino',  
    items: [  
      { beverage: 'mocaccino', quantity: 1},  
      { beverage: 'expresso', quantity: 2},  
      { beverage: 'capuccino', quantity: 1}  
    ],  
    expectedTotalPrice: 7.30  
  }  
].forEach(scenarioOrderContainsBeverages);
```

Now, we need to change the tests on the edit, quantity, and delete items. Since we have three items, we should check for all of them. We could implement some kind of loop to create a promise for each assertion about each item action. Then, we could use Q.all to wait for all the assertions to be fulfilled or for at least one to be rejected. However, this is actually not a good idea. The resulting code would be complex; in our tests, we should favor code that is simple and expressive. After all, it is just a test, not production code.

What we actually need is a separate test for each separate assertion. Until now, we have had all the assertions for all the edit quantity actions in a single test. We are asserting for the first and second items in the same test. The same thing applies to the remove item action.

Let's change our tests:

```
testExample.items.forEach(function (itemExample, i) {  
  
    it('will be possible to remove the ' + itemExample.beverage,  
        function () {  
            return expect(this.result).to.eventually  
                .have.property('actions')  
                .that.include(this.orderActions.removeItem(i));  
        };  
  
    it('will be possible to change the quantity of ' +  
        itemExample.beverage, function () {  
        return expect(this.result).to.eventually  
            .have.property('actions')  
            .that.include(this.orderActions.editItemQuantity(i));  
    };  
});
```

This is much better. Now, we have a different test for each item. We can even have a much better and explicit title for each test; this will produce a much informative report.

Do not abuse these techniques. It is more important for the tests to be easily readable than to save a bunch of lines of code. As a rule of thumb, it is OK to parameterize one scenario if we conserve exactly the same set of assertions for each test data example. For example, one can be tempted to merge the "empty order" with the "nonempty order" scenario. After all, it is just different setup data, right? However, if we do this, then we need to parameterize somehow whether the place order is expected to be available or not (an empty order should not be). The same thing applies to testing: there are no other actions available, except for adding an item. Maybe, we need to change the title of the tests, depending on whether the order is empty or not, to offer a clearer test report. This is too complicated, and the resulting test code would be less readable, so it is better not to do it. As always, just use your common sense to decide whether it is reasonable to parameterize one scenario or not.

Finishing our feature

We are almost done; we just need to finish our last scenario. First of all, we need a new DAO that is responsible for storing the messages. The real implementation of this DAO, perhaps, will not go to the database but will probably use the flash scope or the session scope of the web framework we are going to use. This does not matter conceptually as it is an object to access external data:

```
beforeEach(function () {
    this.orderStorage = newStorage();
    this.messageStorage = newStorage();
    this.orderSystem = orderSystemWith({
        order: this.orderStorage.dao(),
        message: this.messageStorage.dao()
    });
});
```

Now, we can create an example message factory in the `test/support/examples/errors.js` file:

```
'use strict';

module.exports = {
    badQuantity: function (quantity) {
        return {
            key: "error.quantity",
            params: [quantity]
        };
    },
    beverageDoesNotExist: function () {
        return {
            key: "error.beverage.notExists"
        };
    }
};
```

For now, we can imagine two errors: when the user tries to order a quantity that is not a number or is less than one and when the requested beverage does not exist.

Now we can change our tests:

```
var chai = require('chai'),
    expect = chai.expect,
    newStorage = require('../support/storageDouble'),
```

```
order = require('./support/examples/orders'),
errors = require('./support/examples/errors'),
orderSystemWith = require('../lib/orders');

chai.use(require("chai-as-promised"));

describe('Customer displays order', function () {

  beforeEach(function () {
    this.orderStorage = newStorage();
    this.messageStorage = newStorage();
    this.orderSystem = orderSystemWith({
      order: this.orderStorage.dao(),
      message: this.messageStorage.dao()
    });
  });

  context('Given that the order is empty', function () {
    beforeEach(function () {
      this.order = this.orderStorage.alreadyContains(order.empty());
      this.messages = this.messageStorage.alreadyContains({
        id: this.order.id,
        data: []
      });

      this.result = this.orderSystem.display(this.order.id);
    });
    // Skipped for brevity
  });

  function scenarioOrderContainsBeverages(testExample) {
    context('Given that the order contains ' + testExample.title,
    function () {
      beforeEach(function () {
        this.order = this.orderStorage.alreadyContains(order.
withItems(testExample.items));
        this.messages = this.messageStorage.alreadyContains({
          id: this.order.id,
          data: []
        });
        this.orderActions = order.actionsFor(this.order);
      });
    });
  }
});
```

```
        this.result = this.orderSystem.display(this.order.id);
    });
    // Skipped for brevity
});
}
// Skipped for brevity
context('Given that the order has pending messages', function () {
    beforeEach(function () {
        this.order = this.orderStorage.alreadyContains(order.empty());
        this.messages = this.messageStorage.alreadyContains({
            id: this.order.id,
            data: [errors.badQuantity(-1)]
        });

        this.result = this.orderSystem.display(this.order.id);
    });
    it('will show the pending messages', function () {
        return expect(this.result).to.eventually
            .have.property('messages')
            .that.is.deep.equal(this.messages.data);
    });
    it('there will be no more pending messages');
});
});
```

We need to add the setup for the new DAO in the other scenarios too. The new test itself is very simple.

To implement the remaining test, we need to think about what needs to happen so that the messages are not shown again. The answer is simple: we need to test whether we are updating the messages to an empty array after displaying them. We need to change the `storageDouble.js` file to add a method to update:

```
module.exports = function () {
    var dao = {
        byId: sinon.stub(),
        update: sinon.stub()
    },
    storage = {};
storage.updateWillNotFail= function() {
    dao.update.callsArgWithAsync(1, null);
};
// Skipped for brevity
};
```

We not only need to add the update method, but also a setup method to ensure that the update will be performed successfully. Otherwise, the test double will never call the callback, and our test will time-out.



Note that *not* all the doubles in our system need to have the same interface or be DAOs. If you really think that `messageDAO` should have another interface, just create a new kind of test double.

Now, we just need to write our test:

```
var chai = require('chai'),
    expect = chai.expect,
    newStorage = require('../support/storageDouble'),
    order = require('../support/examples/orders'),
    errors = require('../support/examples/errors'),
    orderSystemWith = require('../lib/orders');

chai.use(require("sinon-chai"));
chai.use(require("chai-as-promised"));

describe('Customer displays order', function () {
  // Skipped for brevity
  context('Given that the order is empty', function () {
    beforeEach(function () {
      this.order = this.orderStorage.alreadyContains(order.empty());
      this.messages = this.messageStorage.alreadyContains({
        id: this.order.id,
        data: []
      });
      this.messageStorage.updateWillNotFail();
    });

    this.result = this.orderSystem.display(this.order.id);
  });
  // Skipped for brevity
});

function scenarioOrderContainsBeverages(testExample) {
  context('Given that the order contains ' + testExample.title,
  function () {
    beforeEach(function () {
      this.order = this.orderStorage.alreadyContains(order.
withItems(testExample.items));
    });
  });
}
```

```
        this.messages = this.messageStorage.alreadyContains({
            id: this.order.id,
            data: []
        });
        this.messageStorage.updateWillNotFail();
        this.orderActions = order.actionsFor(this.order);

        this.result = this.orderSystem.display(this.order.id);
    });
    // Skipped for brevity
});
}
// Skipped for brevity
context('Given that the order has pending messages', function () {

    beforeEach(function () {
        this.order = this.orderStorage.alreadyContains(order.empty());
        this.messages = this.messageStorage.alreadyContains({
            id: this.order.id,
            data: [errors.badQuantity(-1)]
        });
        this.messageStorage.updateWillNotFail();

        this.result = this.orderSystem.display(this.order.id);
    });

    it('will show the pending messages', function () {
        return expect(this.result).to.eventually
            .have.property('messages')
            .that.is.deep.equal(this.messages.data)
    });

    it('there will be no more pending messages', function () {
        var dao = this.messageStorage.dao(),
            orderId = this.order.id;
        return this.result.then(function () {
            expect(dao.update)
                .to.be.calledWith({
                    id: orderId,
                    data: []
                });
        });
    });
});
});
```

We need to explicitly state in the setup that the update operation of the message will not fail. This is nice, as it reminds us that we should write an additional scenario about handling failure.

The main problem now is that the test is a bit convoluted, because we need to wait for the operation to finish before actually making the assertion against the test double. We can solve this issue by simply waiting for the operation to complete in the `beforeEach` block:

```
context('Given that the order has pending messages', function () {
  beforeEach(function () {
    this.order = this.orderStorage.alreadyContains(order.empty());
    this.messages = this.messageStorage.alreadyContains({
      id: this.order.id,
      data: [errors.badQuantity(-1)]
    });
    this.messageStorage.updateWillNotFail();

    this.result = this.orderSystem.display(this.order.id);

    return this.result;
  });

  it('will show the pending messages', function () {
    return expect(this.result).to.eventually
      .have.property('messages')
      .that.is.deep.equal(this.messages.data)
  });

  it('there will be no more pending messages', function () {
    expect(this.messageStorage.dao().update)
      .to.be.calledWith({
        id: this.order.id,
        data: []
      });
  });
});
```

We can make it a bit nicer still if we create a method that represents the assertion in the test object itself:

```
var chai = require('chai'),
  expect = chai.expect,
  sinon = require('sinon');
```

```
module.exports = function () {
  var dao = {
    getById: sinon.stub(),
    update: sinon.stub()
  },
  storage = {};
// Skipped for brevity
storage.toExpectUpdate = function (entity) {
  expect(dao.update).to.be.calledWith(entity);
};

return storage;
};
```

Now, our test can be rewritten as follows:

```
it('there will be no more pending messages', function () {
  this.messageStorage.toExpectUpdate({
    id: this.order.id,
    data: []
  });
});
```

Finally, we just need to parameterize the scenario:

```
function scenarioOrderHasPendingMessages(testExample) {
  context('Given that the order has pending the following messages: ' +
+ testExample.title, function () {
  beforeEach(function () {
    this.order = this.orderStorage.alreadyContains(order.empty());
    this.messages = this.messageStorage.alreadyContains({
      id: this.order.id,
      data: testExample.pendingMessages
    });
    this.messageStorage.updateWillNotFail();

    return this.result = this.orderSystem.display(this.order.id);
  });
  // Skipped for brevity
});
}

[
  {
    title: 'bad quantity[-1]',
    pendingMessages: [errors.badQuantity(-1)]
  },
  {
```

```
title: 'beverage does not exist, bad quantity[0]',  
pendingMessages: [  
    errors.beverageDoesNotExist(),  
    errors.badQuantity(-1)  
]  
}  
].forEach(scenarioOrderHasPendingMessages);
```

Summary

In this chapter, you learned some of the following tricks to write better features:

- Decompose the system into the principal entities. Then analyze which actions each role can perform on each entity. For each action you discover, there must be a feature.
- Do not forget that the act of displaying or showing information is an action itself, so it needs its own feature.
- Decompose a feature in a set of scenarios where each scenario represents a different setup or input data that will lead to a different outcome.
- We need a specific test for each outcome of each scenario.
- The returned data is not the only possible outcome, but there can be side effects and available actions.
- Test an asynchronous system based either on callbacks or promises.
- Build asynchronous test doubles using either callbacks or promises.
- Test doubles should faithfully resemble the technology we use to access external systems, databases, and other features. This includes not only the interface, but also the resulting data schemas.
- Make a more efficient setup that extracts the example test-data creation to example factory objects.
- Create smart test objects that represent external systems and that will encapsulate all the test double creation, setup, and assertion logic.
- Build parameterized scenarios.

In the next chapter, you will learn another tool used for BDD: Cucumber.js. We will learn how to express features in a more human-friendly format used by Cucumber.js: Gherkin. Finally, we will implement another feature of myCafé and see how we can isolate the testing of both features and decouple them.

4

Cucumber.js and Gherkin

In the last two chapters we covered how to effectively use Mocha from the point of view of BDD, but Mocha is not the only option we have. There are plenty of test runners that we could have used to implement our BDD tests. Other tools such as Vows (<http://vowsjs.org/>) work perfectly well (some claim even better than Mocha) to do BDD. However, I have chosen Mocha because it is the most popular tool and because all the things that you have learned with it can be easily applied to other tools. Nonetheless, I would not consider this book complete if I did not show you another tool for comparison purposes, and this tool is Cucumber.js.

In this chapter:

- We will explore how to write BDD features using Gherkin
- You will learn how to automate features written in Gherkin using Cucumber.js
- We will write several helper codes that will simplify this automation
- You will learn the World object pattern used in Cucumber.js
- We will migrate the feature we wrote in the last chapter to Gherkin and Cucumber.js
- We will see how to reuse most of the code of the last chapter to implement tests using Cucumber.js

Getting started with Gherkin and Cucumber.js

Gherkin is a domain-specific language, specialized in describing features and scenarios. The point of Gherkin is that it is extremely similar to natural language, so similar that domain experts can read and understand features written in Gherkin or, in some lucky cases, write them themselves.

This is a very important advantage, since BDD aims not only to automate tests, but also to improve communication between domain experts and software developers.

To introduce the Gherkin language, we will migrate to this language the "display order" feature we coded in the previous chapter, and then we will automate it using Cucumber.js.

For brevity, I will not show system error scenarios. Keep in mind that, if you were writing a real application, you would need to add scenarios about these kinds of errors. You need to address things such as what happens when one of the external systems, such as the orders database, is down!

Preparing your project

We can prepare a project from scratch but, for this example, it is faster to just reuse the project we already have. Simply make a copy of the myCafé project, and then you can remove the `test/customer_displays_order.js` file but keep the `test/support/` folder. Remove the `node_modules/` folder and edit the `package.json` file to remove all the references to Mocha. It should look similar to this:

```
{  
  "name": "mycafe",  
  "version": "0.1.0",  
  "description": "A sample app for BDD with JS",  
  "main": "index.js",  
  "author": "Enrique Amodeo",  
  "license": "MIT",  
  "devDependencies": {  
    "chai": "^1.9.1",  
    "chai-as-promised": "^4.1.1",  
    "sinon": "^1.10.3",  
    "sinon-chai": "^2.5.0"  
  },  
}
```

```

  "dependencies": {
    "q": "^1.0.1"
  }
}

```

Now you should execute `npm install` to reinstall the remaining dependencies. Then just install the `cucumber` package:

```
$ me@~/mycafe> npm install --save-dev cucumber
```

After the installation, we can edit the `package.json` file to add a test script:

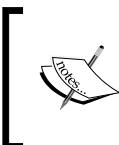
```

{
  "name": "mycafe",
  "version": "0.1.0",
  "description": "A sample app for BDD with JS",
  "main": "index.js",
  "scripts": {
    "test": "cucumber.js --format pretty"
  },
  "author": "Enrique Amodeo",
  "license": "MIT",
  "devDependencies": {
    "chai": "^1.9.1",
    "chai-as-promised": "^4.1.1",
    "cucumber": "^0.4.1",

    "sinon": "^1.10.3",
    "sinon-chai": "^2.5.0"
  },
  "dependencies": {
    "q": "^1.0.1"
  }
}

```

The test script will just invoke the command-line tool of Cucumber.js, called `cucumber.js`, and tell it to use the `pretty` reporter.



If you are using Windows, then you must use `cucumber-js` instead of `cucumber.js` as the executable name. Fortunately, this detail will be hidden inside our `package.json` file, and we will be able to run Cucumber.js simply using `npm test`.

Let's start writing our first scenario!

Writing your first scenario in Gherkin

We can create a folder called `features/` in the root of the new project. In general, inside the `features/` folder, we will create an additional folder for each use case, subsystem, or business process that our project implements. In this example, we will create a `making_an_order/` folder, and inside it we will add `customer_displays_order.feature` with the following contents:

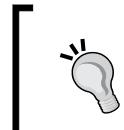
```
Feature: Customer displays order

Part of the "Making an Order" epic

As a Customer
I want to display the order
in order to review the contents of my order and its price easily
```

Nothing really impressive, but we have created our first Gherkin file. All Gherkin files start with the `Feature:` keyword followed by the title of the feature. As a title, we can use exactly the same title we used for our Mocha feature. The title of the feature will be used in the test report.

The following paragraphs are simple, free text that describes the feature. This description has no effect from the test automation point of view, but it is helpful as general documentation.



As a description, I often use the canonical form of a user story:
As a <ROLE> I want to <ACTION> in order to <VALUE FOR THE USER>. This usually plays very well with the BDD approach and with agile teams in general.

Now, we can try to add our first scenario to the feature:

```
Feature: Customer displays order

Part of the "Making an Order" epic

As a Customer
I want to display the order
in order to review the contents of my order and its price easily

Scenario: Order is empty
  Given that the order is empty
```

```
When the customer displays the order
Then no order items will be shown
And "0" will be shown as total price
And there will only be possible to add a beverage
```

Every scenario must start with the `Scenario:` keyword. Optionally, we can put a title to each scenario. As we saw in the last chapter, we should indicate in the title the circumstance in the setup, or in the input data, that leads to a different outcome for the same action.

The actual content of the scenario is defined as a series of steps. A step in Gherkin is any sentence that starts with one of the following keywords: `Given`, `When`, `Then`, `And`, and `But`. Actually, these keywords have no effect on the automation of the tests, beyond signaling the beginning of each step. However, there is a rather strict style about using them, which is as follows:

- Use the `Given` keyword to start a step intended to perform a piece of the setup of our scenario. A setup step should be worded in the past or in the present, specifying the state of the system before executing the scenario action.
- Use the `When` keyword to specify the feature action in the present tense.
- Use the `Then` keyword to specify any outcome of the action. Each of these steps represents a specific test or assertion.
- The `But` and `And` keywords are neutral and can be used whenever we want to make our scenario more expressive. However, it is bad style to use them in the first step of the setup or in the first assertion.

The structure of a scenario follows a `Given/When/Then` pattern. In the `Given` steps, we define the setup of the test. In the `When` step, we define the action that this feature is describing. There must be only one `When` step, since all features describe a single action on the system, but the tool will not enforce this. The `Then` steps define the tests or assertions we want to perform.

Note the resemblance between this structure and the one we used in our Mocha contexts. As a rule of thumb, for each Mocha context, you should have a similar Gherkin scenario. We have changed the wording a little to make the scenario more readable, but the titles of the tests and contexts correspond almost exactly with the steps and scenarios.



You can comment a single line in Gherkin using the `#` character.
There are no multiline comments.



Executing Gherkin

Now that we have a feature with a scenario, we can execute it issuing the `npm test` command in the command line. The result looks something like this:

```
Feature: Customer displays order
  Part of the "Making an Order" epic

  As a Customer
  I want to display the order
  in order to review the contents of my order and its price easily

  Scenario: Order is empty
    Given that the order is empty                               # features/making_an_order/customer_displays_order.feature:9
    When the customer displays the order                         # features/making_an_order/customer_displays_order.feature:10
    Then no order items will be shown                          # features/making_an_order/customer_displays_order.feature:11
    And "0" will be shown as total price                      # features/making_an_order/customer_displays_order.feature:12
    And there will only be possible to add a beverage        # features/making_an_order/customer_displays_order.feature:13
                                                          # features/making_an_order/customer_displays_order.feature:14

1 scenario (1 undefined)
5 steps (5 undefined)

You can implement step definitions for undefined steps with these snippets:

this.Given(/^that the order is empty$/, function (callback) {
  // Write code here that turns the phrase above into concrete actions
  callback.pending();
});

this.When(/^the customer displays the order$/, function (callback) {
  // Write code here that turns the phrase above into concrete actions
  callback.pending();
});

this.Then(/^no order items will be shown$/, function (callback) {
  // Write code here that turns the phrase above into concrete actions
  callback.pending();
});

this.Then(/^"([^\"]*)" will be shown as total price$/, function (arg1, callback) {
  // Write code here that turns the phrase above into concrete actions
  callback.pending();
});

this.Then(/^there will only be possible to add a beverage$/, function (callback) {
  // Write code here that turns the phrase above into concrete actions
  callback.pending();
});
```

Executing Gherkin with Cucumber without code!

Wow! We did not write any JavaScript, and Cucumber is able to offer us an understandable report. It tells us that all the steps are undefined and that we should automate them somehow. For this, Cucumber.js offers us some code examples that we could use to automate step execution.

The next step is to create a new JavaScript file, `test/step_definitions/display_order_steps.js`, with the following contents:

```
'use strict';

module.exports = function () {
  this.Given(/^that the order is empty$/, function (cb) {
    cb.pending();
```

```

}) ;

this.When(/^the customer displays the order$/, function (cb) {
  cb.pending();
});

this.Then(/^no order items will be shown$/, function (cb) {
  cb.pending();
});

this.Then(/^"( [^"]*)" will be shown as total price$/, function
(expectedTotalPrice, cb) {
  cb.pending();
});

this.Then(/^there will only be possible to add a beverage$/, function
(cb) {
  cb.pending();
});
};

```

We just copied what Cucumber.js proposed, changing some parameter names. When Cucumber.js loads this module, it will execute the exported function using an instance of the Cucumber.js API as a value for `this`. This way, we can access the Cucumber.js API using the `this` keyword. Using the `Given`, `When`, and `Then` methods, we can register handlers for the different steps we wrote in our scenarios. All of these functions are aliases of each other and perform exactly the same task: registering a function that will handle the execution of the step that matches a certain regular expression.

If you think that repeating `this` in `this.Given`, `this.When`, and `this.Then` is too verbose, there is a nice trick: you can just assign them to local variables and go without `this`. Have a look at the following code:



```

module.exports = function() {
  var Given = this.Given,
      When = this.When,
      Then = this.Then;

  Given(/^some setup step$/, function(cb) {...});
  When(/^some action step$/, function(cb) {...});
  Then(/^some assertion step$/, function(cb) {...});
}

```

When Cucumber.js is executed, it will just parse the Gherkin of our features and look for the registered handler whose regular expression has the best match for the step. If it finds one, it will execute the handler function using the extracted parameters from the regular expression.



It is a good practice to enclose the parameters in quotes. This makes it easier to capture them in the regular expression. This is exactly what "`([^"]*)`" does: capture all the characters between double quotes that are not double quotes themselves.

The handler function always receives a callback function as its last parameter. When Cucumber.js executes our handler function, it waits until we do one of the following things:

- The handler function executes the callback without parameters or with a *falsy* parameter to indicate that the step is executed without errors.
- The handler function executes the callback with a *truthy* parameter, indicating that there was an error or an assertion failed. Alternatively, we can invoke `cb.fail(error)`, which is a bit more verbose but more descriptive.
- The handler function executes `cb.pending()`, indicating that the step is not yet ready to be implemented properly.



A problem with the Cucumber.js approach is that we are forced to call the callback, even if our step handler is totally synchronous. So do not forget to invoke the callback!

When you register a step handler function using `Given`, `When`, or `Then`, you can use either a regular expression or a string. For example, consider the following step:

```
this.Then(/"(.*?)"/ will be shown as total price$/, function  
  (expectedTotalPrice, cb) {  
    cb.pending();  
  });
```

The preceding step can be rewritten as:

```
this.Then('"$price" will be shown as total price', function  
  (expectedTotalPrice, cb) {  
    cb.pending();  
  });
```

The string style is simpler than the regular expression, so unless we really need the full power of the regular expressions it is better to stick to the string style. In the string style, we simply use double quotes to get a parameter. With a regular expression, we need to use a capture group.

If you execute `npm test` again, you will see that nothing has changed. Do not worry; we just need to tell Cucumber.js the location of the JavaScript files with the code that automates the steps. We just need to change `package.json`:

```
"scripts": {
  "test": "cucumber.js --format pretty --require
    test/step_definitions/"
},
```

The `--require` parameter will tell Cucumber.js to include all the JavaScript modules in the `test/step_definitions/` folder. This way, it will see the implementation of the steps. If you execute the tests again, you will see the following output:

```
$ npm test
> mycafe@0.1.0 test /Users/ham/mycafe
> cucumber.js --format pretty --require test/step_definitions/

Feature: Customer displays order
  Part of the "Making an Order" epic

  As a Customer
  I want to display the order
  in order to review the contents of my order and its price easily

  Scenario: Order is empty
    Given that the order is empty
    When the customer displays the order
    Then no order items will be shown
    And "0" will be shown as total price
    And there will only be possible to add a beverage
```

features/making_an_order/customer_displays_order.feature:9
features/making_an_order/customer_displays_order.feature:10
features/making_an_order/customer_displays_order.feature:11
features/making_an_order/customer_displays_order.feature:12
features/making_an_order/customer_displays_order.feature:13
features/making_an_order/customer_displays_order.feature:14

```
1 scenario (1 pending)
5 steps (1 pending, 4 skipped)
```

Steps present, but still pending!

Now, we can see that it found all the step handlers, and it tried to execute the scenario. When it executed the first step, the handler function called `cb.pending()`, marking this step as pending. In this case, Cucumber.js will not continue to execute the rest of the steps of the scenario and will mark them as skipped.



You can have your steps handlers in several files; there is no need to put them all in the same file. In the future you could end up with other files, such as `test/step_definitions/add_beverages_to_order_steps.js` or `test/step_definitions/common_steps.js`.

It is now time to start adding real code to our steps handlers. For this, we will use the `World` object pattern, which is a recommended best practice in the Cucumber ecosystem.

The World object pattern

In Cucumber.js, all the step handler functions are executed with a special object, called the `World`, as a runtime context. So, whenever we reference `this` inside a function handler, we access the `World` object. The `World` object has two interesting properties:

- A new instance of the `World` object is created before each scenario. We can safely share information between steps through the `World` object, without any fear of mixing states between scenarios. So, the best practice is to store any state of the current scenario in the `World` object.
- We can create our custom `World` object. This is very helpful because we can add all the utilities and helpers that our steps need.

Since the `World` object is used at the same time as a test support API, we will build our own on top of the support code we used in the previous chapter. Let's create a `test/support/world.js` file with the following contents:

```
'use strict';

var newStorage = require('./storageDouble'),
    orderSystemWith = require(' ../../lib/orders');

module.exports = function (cb) {
  this.orderStorage = newStorage();
  this.messageStorage = newStorage();
  this.orderSystem = orderSystemWith({
    order: this.orderStorage.dao(),
    message: this.messageStorage.dao()
  });

  cb(); // We are done!
};


```

We simply exported a function that will be used by Cucumber.js to construct a brand new *World* object. Cucumber.js expects this function to be asynchronous, even if it is not the case, so it will pass us a callback that we must call to mark the world construction as done.

Since a new *World* instance will be created for each scenario, we simply put here the code we were using in the last chapter to initialize our order system and its test doubles.

There is an alternative way of constructing a *World* object:

```
module.exports = function (cb) {
  var world = {};
  world.orderStorage = newStorage();
  world.messageStorage = newStorage();
  world.orderSystem = orderSystemWith({
    order: world.orderStorage.dao(),
    message: world.messageStorage.dao()
  });

  cb(world); // We are done!
};
```

In this style, we can create an object and simply pass it as an argument to the callback. If we invoke the callback without arguments, Cucumber.js will use the value of `this` as the new *World* instance. If we pass a parameter, it will become the *World* instance. Which style to use? It depends on your personal style of JavaScript coding!

Now that we have the world in place, we can start implementing our step handlers. The first thing is to tell Cucumber.js that it must use our *World* implementation:

```
var chai = require('chai'),
  expect = chai.expect,
  order = require('../support/examples/orders');

chai.use(require("sinon-chai"));
chai.use(require("chai-as-promised"));

module.exports = function () {
  this.World = require("../support/world.js");
  // Skipped tests for brevity
};
```

Apart from importing Chai and our examples factory, we will import our world module and assign it to `this.World`. Cucumber.js will use whatever function is in `this.World` as the world object factory. Note that Cucumber.js requires the name of this factory to be `this.World`.

Let's now implement the Given step:

```
this.Given('that the order is empty', function (cb) {
  this.order = this.orderStorage.alreadyContains(order.empty());
  this.messages = this.messageStorage.alreadyContains({
    id: this.order.id,
    data: []
  });
  this.messageStorage.updateWillNotFail();
  cb();
});
```

That is exactly the same code we had in the `beforeEach` block of the empty order scenario in the previous chapter. The only difference is that we need to call the callback, even if our setup is synchronous.

Let's go for the When step:

```
this.When('the customer displays the order', function (cb) {
  this.result = this.orderSystem.display(this.order.id);
  cb();
});
```

Nothing new; again, we will call the callback immediately. Since we are returning a promise, we do not need to wait for it to be resolved in the When step. We will wait for it using `chai-as-promised` in the Then steps:

```
this.Then('no order items will be shown', function (cb) {
  expect(this.result).to.eventually
    .have.property('items').that.is.empty
    .then(function (ignoredItems) {
      cb();
    }, cb);
});
```

So, we implemented our first assertion. This is a bit weird! Of course, we must wait for the eventual assertion made by `chai-as-promised` to finish, but the following code should have worked:

```
this.Then('no order items will be shown', function (cb) {
  expect(this.result).to.eventually
    .have.property('items').that.is.empty
    .then(cb, cb);
});
```

We wait for the promise to end using the `then` method, passing the callback as both parameters. If the assertion passes, then the callback will be called, since we pass it as the success handler. If the assertion fails, then the callback is called with an error, and Cucumber.js will fail. Why is the preceding code not correct? The problem is that the promise returned by `chai-as-promised` will pass the `result` parameter to the success handler when it is fulfilled successfully. This will make Cucumber.js fail, so we need to explicitly ignore the result.



If you are wondering about the value returned by a `sinon-as-promised` promise when the assertion is successful, the answer is the actual value you are testing against in the assertion.

With this weird caveat, we can implement the rest of the steps:

```
this.Then('$price" will be shown as total price', function
(expectedTotalPrice, cb) {
  expect(this.result).to.eventually
    .have.property('totalPrice')
    .that.is.equal(Number(expectedTotalPrice))
    .then(function (ignored) {
      cb();
    }, cb);
});

this.Then('there will only be possible to add a beverage', function
(cb) {
  expect(this.result).to.eventually
    .have.property('actions')
    .that.is.deep.equal([
      {
```

Cucumber.js and Gherkin

```
        action: 'append-beverage',
        target: this.order.id,
        parameters: {
          beverageRef: null,
          quantity: 0
        }
      ]
    )
  .then(function (ignored) {
    cb();
  }, cb);
}) ;
```

Just notice the code in the test about the price. Yes, Cucumber.js will extract the parameters using regular expressions, but they are always strings! We need to cast it to a number.

Now you can execute the tests and see the scenario passing (assuming that you implemented the necessary production code in the previous chapter):

```
$ npm test
> mycafe@0.1.0 test /Users/iham/mycafe
> cucumber.js --format pretty --require test/step_definitions/

Feature: Customer displays order
  Part of the "Making an Order" epic
  As a Customer
  I want to display the order
  in order to review the contents of my order and its price easily

  Scenario: Order is empty
    Given that the order is empty          # features/making_an_order/customer_displays_order.feature:9
    When the customer displays the order   # features/making_an_order/customer_displays_order.feature:10
    Then no order items will be shown     # features/making_an_order/customer_displays_order.feature:11
    And "0" will be shown as total price  # features/making_an_order/customer_displays_order.feature:12
    And there will only be possible to add a beverage # features/making_an_order/customer_displays_order.feature:13
                                                # features/making_an_order/customer_displays_order.feature:14

1 scenario (1 passed)
5 steps (5 passed)
```

A great success!

Now, everything is green!

Better step handlers

So far, the code is a bit ugly, since we need to mess with callbacks and promises. Unfortunately, unlike Mocha, Cucumber.js does not offer an easy way to deal with promises. However, we can create a small utility for this. Create a file called `test/support/cucumber_sugar.js` with the following contents:

```
'use strict';

module.exports = function (stepHandler) {
  return function () {
    var cb = arguments[arguments.length - 1];

    try {
      var result = stepHandler.apply(this, arguments);

      if (result && typeof result.then === 'function') {
        result.then(function (ignoredParam) {
          cb()
        }, cb);
      } else
        cb();
    } catch (err) {
      cb(err);
    }
  };
};
```

This code is a bit advanced, but it mainly takes a step handler function that is written according to the Mocha spec and transforms it into a normal Cucumber.js step handler. Basically, it executes the provided function with the same arguments that it receives and inspects the result. If the result has a `then` method, it considers that the step handler has returned a promise, and we just attach the Cucumber.js callback to the promise. If not, we just call the callback to tell Cucumber.js that the step handler is finished.

We can change our steps to use this utility:

```
'use strict';

var chai = require('chai'),
  expect = chai.expect,
```

Cucumber.js and Gherkin

```
order = require('../support/examples/orders'),
sugar = require('../support/cucumber_sugar');

chai.use(require("sinon-chai"));
chai.use(require("chai-as-promised"));

module.exports = function () {
  this.World = require("../support/world.js");

  this.Given('that the order is empty', sugar(function () {
    this.order = this.orderStorage.alreadyContains(order.empty());
    this.messages = this.messageStorage.alreadyContains({
      id: this.order.id,
      data: []
    });
    this.messageStorage.updateWillNotFail();
  }));

  this.When('the customer displays the order', sugar(function () {
    this.result = this.orderSystem.display(this.order.id);
  }));

  this.Then('no order items will be shown', sugar(function () {
    return expect(this.result).to.eventually
      .have.property('items').that.is.empty;
  }));

  this.Then('"$price" will be shown as total price', sugar(function (expectedTotalPrice) {
    return expect(this.result).to.eventually
      .have.property('totalPrice')
      .that.is.equal(Number(expectedTotalPrice));
  }));

  this.Then('there will only be possible to add a beverage',
sugar(function () {
  return expect(this.result).to.eventually
    .have.property('actions')
    .that.is.deep.equal([
      {
        action: 'append-beverage',
        target: this.order.id,
        parameters: {

```

```

        beverageRef: null,
        quantity: 0
    }
}
]);
}) );
};

```

We only need to wrap our step's handlers using the `sugar` utility method. Now our steps look exactly as in Mocha and are much more readable!

Better reporting

Another problem is the error reporting. If an assertion fails, the Cucumber.js reporter will not offer a diff of the expected and actual values if the assertion of Chai is using the `deep` flag. The real problem is that the Cucumber.js reporter does not know how to interpret the Chai assertion error and ignore the `showDiff`, `actual`, and `expected` fields. If we had an error in our code, then the report error would be like this:

```

Scenario: Order is empty
  Given that the order is empty
  When the customer displays the order
  Then no order items will be shown
  And "0" will be shown as total price
  And there will only be possible to add a beverage
    # features/making_an_order/customer_displays_order.feature:9
    # features/making_an_order/customer_displays_order.feature:10
    # features/making_an_order/customer_displays_order.feature:11
    # features/making_an_order/customer_displays_order.feature:12
    # features/making_an_order/customer_displays_order.feature:13
    # features/making_an_order/customer_displays_order.feature:14
AssertionError: expected [ Array(1) ] to deeply equal [ Array(1) ]
  at [Object Object].assertEqual (/Users/iham/mycafe/node_modules/chai/lib/chai/core/assertions.js:393:19)
  at [Object Object].ctx.(anonymous function) (/Users/iham/mycafe/node_modules/chai/lib/chai/utils/addMethod.js:40:25)
  at /Users/iham/mycafe/node_modules/chai-as-promised/lib/chai-as-promised.js:302:26
  at _fulfilled (/Users/iham/mycafe/node_modules/q/q.js:787:54)
  at self.promisedDispatch.done (/Users/iham/mycafe/node_modules/q/q.js:816:30)
  at Promise.promiseDispatch (/Users/iham/mycafe/node_modules/q/q.js:749:13)
  at /Users/iham/mycafe/node_modules/q/q.js:557:44
  at flush (/Users/iham/mycafe/node_modules/q/q.js:108:17)
  at process._tickCallback (node.js:419:13)
  at Function.Module.runMain (module.js:499:11)
  at startup (node.js:119:16)
  at node.js:906:3

```

Not a very useful error message

To solve this, we can write our own Cucumber.js reporter, but this is expensive. We can go with a cheaper solution:

```

function simpleDiffReport(cb) {
  return function (err) {
    if (err) {
      if (typeof err.expected !== 'undefined' &&
          typeof err.actual !== 'undefined') {
        var errMsg = [];
        errMsg.push('Expected:');
        errMsg.push(JSON.stringify(err.expected));
        errMsg.push('Actual:');

```

```
        errMsg.push(JSON.stringify(err.actual));
        errMsg.push(err.stack);
        cb(errMsg.join('\r\n'));
    } else
        cb(err);
} else
    cb();
};

}

module.exports = function (stepHandler) {
    return function () {
        var cb = simpleDiffReport(arguments[arguments.length - 1]);
        // Skipped for brevity
    };
}
```

We simply wrapped the callback in a function that will inspect the Chai assertion error to compose a more meaningful message. With this change, our error message now looks like this:

```
Scenario: Order is empty
  Given that the order is empty
  When the customer displays the order
  Then no order items will be shown
  And "0" will be shown as total price
  And there will only be possible to add a beverage # features/making_an_order/customer_displays_order.feature:14
    Expected:
      [{"action": "append-beverage", "target": "<empty order>", "parameters": {"beverageRef": null, "quantity": 0}}]
    Actual:
      [{"action": "append-beverage", "target": "<empty order>", "parameters": {"beverageRef": null, "quantity": 10}}]
AssertionError: expected [ Array(1) ] to deeply equal [ Array(1) ]
  at [object Object].assertEqual (/Users/iham/mycafe/node_modules/chai/lib/chai/core/assertions.js:393:19)
  at [object Object].ctx.(anonymous function) (/Users/iham/mycafe/node_modules/chai/lib/chai/utils/addMethod.js:40:25)
  at /Users/iham/mycafe/node_modules/chai-as-promised/lib/chai-as-promised.js:302:26
  at _fulfilled (/Users/iham/mycafe/node_modules/q/q.js:787:54)
  at self.promiseDispatch.done (/Users/iham/mycafe/node_modules/q/q.js:816:30)
  at Promise.promiseDispatch (/Users/iham/mycafe/node_modules/q/q.js:749:13)
  at /Users/iham/mycafe/node_modules/q/q.js:557:44
  at flush (/Users/iham/mycafe/node_modules/q/q.js:108:17)
  at process._tickCallback (node.js:419:13)
  at Function.Module.runMain (module.js:499:11)
  at startup (node.js:119:16)
  at node.js:986:3
```

At least we know what we expect and what is returned

This error report is much more informative as we now know that the problem is the quantity field; it is 10, and it should be 0. This will help us to debug the error better.

Writing advanced scenarios

At this point, we have the first scenario working and, thanks to our sugar utility, we have better error reporting and less clutter in our test code. Now, we will try to address the upcoming scenarios that are a bit more complicated.

Gherkin example tables

We could write the following Gherkin code for the scenario about nonempty orders:

```
Feature: Customer displays order
  // Skipped for brevity

  Scenario: Order is empty
  // Skipped for brevity

  Scenario: Non empty order
    Given that the order contains "1" "Expresso"
    And that the order contains "2" "Mocaccino"
    When the customer displays the order
    Then the order will show "1" "Expresso"
    And the order will show "2" "Mocaccino"
    And "6.10" will be shown as total price
    And there will be possible to "place order"
    And there will be possible to "add beverage"
    And there will be possible to "place order"
    And there will be possible to "edit item quantity" for item
      "1"
    And there will be possible to "remove item" for item "1"
    And there will be possible to "edit item quantity" for item
      "2"
    And there will be possible to "remove item" for item "2"
```

The problem with this scenario is that it is very redundant and verbose. One way to solve this is by using Gherkin tables:

```
Scenario: Non empty order
Given that the order contains:
| beverage | quantity |
| Expresso | 1           |
| Mocaccino | 2          |
```

```
When the customer displays the order
Then the following order items are shown:
| beverage | quantity |
| Espresso | 1      |
| Mocaccino | 2      |
And "6.10" will be shown as total price
And there will be possible to:
| action           | for item |
| place order     |          |
| append item     |          |
| edit item quantity | 1      |
| remove item     | 1      |
| edit item quantity | 2      |
| remove item     | 2      |
```

This is much more compact. The tables offer us a way to describe a list of examples that can be consumed by the step handlers. The syntax is quite straightforward; just use the | character to build the cells of the table, and use the first row for the labels.

In the steps, the tables will be available as the first parameter for our step handlers. The Cucumber.js table object has a `hashes()` method that transforms it into an array of plain JavaScript objects. Each one of these objects represents a single row in the table and will contain a field for each column of the table. Each field will have the label of the column as the key and the value of the corresponding cell as the value.

We can now write our handlers for the new steps:

```
this.Given('that the order contains:', sugar(function
(orderItemExamples) {
    this.order = this.orderStorage
        .alreadyContains(order.withItems(orderItemExamples));
    this.messages = this.messageStorage.alreadyContains({
        id: this.order.id,
        data: []
    });
    this.messageStorage.updateWillNotFail();
}));

this.Then('the following order items are shown:', sugar(function
(orderItemExamples) {
    return expect(this.result).to.eventually
        .have.property('items')
```

```

        .that.is.deep.equal(order.items(orderItemExamples));
    });

this.Then('there will be possible to:', sugar(function
(actionExamples) {
    var expectedActions = order
        .actionsForOrderFrom(this.order, actionExamples);

    return expect(this.result).to.eventually
        .have.property('actions')
        .that.have.length(expectedActions.length)
        .and.that.deep.include.members(expectedActions);
}));
```

Although the code is similar to what we did in the last chapter, we have changed `test/support/examples/orders.js` here to be able to deal with the `Cucumber.js` example tables. The new code for this utility is as follows:

```

var beverage = require('./beverages');

var counter = 0;

function asOrderItem(itemExample) {
    return {
        beverage: beverage[itemExample.beverage.toLowerCase()](),
        quantity: Number(itemExample.quantity)
    };
}

function toCamelCase(actionName) {
    return actionName
        .split(/\s+/)
        .map(function (word, i) {
            if (i === 0)
                return word;
            return word.charAt(0).toUpperCase() + word.slice(1);
        })
        .join('');
}

function actionFactoryFor(order) {
    return {
```

```
removeItem: function (index) {
  return {
    action: 'remove-beverage',
    target: order.id,
    parameters: {
      beverageRef: order.data[index].beverage.id
    }
  };
},
editItemQuantity: function (index) {
  var item = order.data[index];
  return {
    action: 'edit-beverage',
    target: order.id,
    parameters: {
      beverageRef: item.beverage.id,
      newQuantity: item.quantity
    }
  };
},
appendItem: function () {
  return {
    action: 'append-beverage',
    target: order.id,
    parameters: {
      beverageRef: null,
      quantity: 0
    }
  };
},
placeOrder: function () {
  return {
    action: 'place-order',
    target: order.id
  };
}
};

module.exports = {
  empty: function () {
    return {

```

```

        id: "<empty order>",
        data: []
    };
},
items: function (itemExamples) {
    return itemExamples.hashes().map(asOrderItem);
},
withItems: function (itemExamples) {
    counter++;
    return {
        id: "<non empty order " + counter + ">",
        data: this.items(itemExamples)
    };
},
actionsForOrderFrom: function (order, actionExamples) {
    var actionFactory = actionFactoryFor(order);

    return actionExamples.hashes().map(function (actionExample) {
        var actionName = toCamelCase(actionExample.action),
            forItem = actionExample['for item'];

        return actionFactory[actionName](Number(forItem) - 1);
    });
}
};

```

We have some changes here. The first one is the auxiliary `asOrderItem` function. It can now handle a single row for an order item example as defined in our Gherkin scenario. It needs to change the beverage name to lowercase and transform the quantity to a number. The exported `items` method calls `hashes()` to obtain a JavaScript array and then maps it to a real array of order items using `asOrderItem`.

The other big change is the `actionsForOrderFrom` method. This will map a table of examples, as described in Gherkin, to the order action objects that we actually need in our code. For this, we will first transform the action name as it comes from the Gherkin to an action name. We used the `toCamelCase` function for this. The old `actionsFrom` public method has been extracted and renamed to a private function called `actionFactoryFor`. This way, we can still create an action factory for a given order and use it in the mapping process. We just need to call the corresponding factory method defined by `actionName`.

I would like to show you the last step handler again:

```
this.Then('there will be possible to:', sugar(function
  (actionExamples) {
    var expectedActions = order
      .actionsForOrderFrom(this.order, actionExamples);

    return expect(this.result).to.eventually
      .have.property('actions')
      .that.have.length(expectedActions.length)
      .and.that.deep.include.members(expectedActions);
  }));
}
```

Note that not only does it use the new `order.actionsForOrderFrom` utility, but it now has a much different assertion from what we had in the last chapter. In the last chapter, we used the `include` assertion for this test, and we created a different test for each action. Now, we defined one step that tests for the whole set of actions, so we need to test for the whole array in one shot.

We might have been tempted to use `to.have.deep.equal` in this case, but this would have been a mistake. Take into account that the order in which our implementation puts the actions into the result array might not be exactly the same order in which the scenario specifies the examples. It is not very sensible to expect both orders to be the same, since this is not really relevant to the good working of our system!

To solve this problem, the assertion gets a bit more complex. Now, we need to use the `deep.include.members` assertion to check whether `expectedActions` is a subset of the `actions` array. This forces us to check whether both arrays have the same length. This will avoid false positives if the resulting set of actions is larger than the expected one and has actions that are not in `expectedActions`.



The `include.members` assertion checks whether all the elements of the provided set are contained in the actual collection, but not the other way around. It does not check for the order of the elements. We can use the `deep` flag to indicate that we want a deep comparison between the elements of both arrays.

Consolidating steps

We can simplify our scenarios if we reword the steps of the empty order scenario a bit:

```
Scenario: Order is empty
  Given that the order contains:
    | beverage | quantity |
  When the customer displays the order
  Then the following order items are shown:
    | beverage | quantity |
  And "0" will be shown as total price
  And there will be possible to:
    | action      |
    | append item |
```

After all, "no items will be shown" is equivalent to showing an empty item table. The same reasoning can be made with regard to the steps about having an order empty or showing only the place order action. If we rephrase this scenario this way, we can simplify our steps in the best possible way – that is, by removing the unused steps! In this case, we can remove the steps for "the order is empty", "no order items will be shown", and "there will only be possible to add a beverage".

The only problem with this approach is that Gherkin is not so terse as it was earlier and looks a bit artificial. So I would like to leave it as it was earlier, but I would like to simplify my steps too! The solution is to write the step handler code once as a reusable function and attach it to all the steps that need it. We can make this change in our code:

```
var theOrderContains = sugar(function (orderItemExamples) {
  this.order = this.orderStorage
    .alreadyContains(order.withItems(orderItemExamples));
  this.messages = this.messageStorage.alreadyContains({
    id: this.order.id,
    data: []
  });
  this.messageStorage.updateWillNotFail();
});
this.Given('that the order contains:', theOrderContains);

var theFollowingItemsAreShown = sugar(function (orderItemExamples) {
  return expect(this.result).to.eventually
    .have.property('items')
```

```
    .that.is.deep.equal(order.items(orderItemExamples));
});

this.Then('the following order items are shown:',
theFollowingItemsAreShown);

var thereWillBePossibleTo = sugar(function (actionExamples) {
  var expectedActions = order.actionsForOrderFrom(this.order,
actionExamples);

  return expect(this.result).to.eventually
    .have.property('actions')
    .that.have.length(expectedActions.length)
    .and.that.deep.include.members(expectedActions);
});
this.Then('there will be possible to:', thereWillBePossibleTo);

this.Given('that the order is empty', function (cb) {
  theOrderContains.call(this, this.dataTable([]), cb);
});

this.Then('no order items will be shown', function (cb) {
  theFollowingItemsAreShown.call(this, this.dataTable([]), cb);
});

this.Then('there will only be possible to add a beverage', function
(cb) {
  thereWillBePossibleTo.call(this, this.dataTable([
    {action: 'append item'}
  ]), cb);
});
```

The step handler function has been given a name, and we can simply from the step registration code (`this.Given`, `this.Then`, and `this.When`). This really does not change anything for those steps. However, now we can invoke them from the other steps and reuse them! There are only two caveats, as follows:

- We need to use the `call(this, param, param....)` syntax to invoke the step handlers. This is because they should run with the runtime context (`this`) that points to the world instance.
- We need to pass the data table examples that they expect. We cannot simply pass an array of objects to the step handler, since they expect a Cucumber.js data table; unfortunately, there is no easy way of constructing one.

To hide the complexity of constructing a Cucumber.js data table, the `dataTable` method has been added to our world object:

```
var newStorage = require('./storageDouble'),
    orderSystemWith = require('../lib/orders'),
    DataTable = require('cucumber').Ast.DataTable,
    Row = DataTable.Row;

module.exports = function (cb) {
  var world = {};
  world.orderStorage = newStorage();
  world.messageStorage = newStorage();
  world.orderSystem = orderSystemWith({
    order: world.orderStorage.dao(),
    message: world.messageStorage.dao()
  });

  world.dataTable = function (dataExamples) {
    var cucumberTable = DataTable();
    if (dataExamples.length === 0)
      return cucumberTable;
    var keys = Object.getOwnPropertyNames(dataExamples[0]);
    cucumberTable.attachRow(Row(keys));
    dataExamples.forEach(function (example) {
      var dataRow = keys.map(function (key) {
        return example[key];
      });
      cucumberTable.attachRow(Row(dataRow));
    });
    return cucumberTable;
  };

  cb(world); // We are done!
};


```

The first thing to do is to import the `Cucumber.Ast.DataTable` and `Cucumber.Ast.DataTable.Row` factory functions. A Cucumber.js `DataTable` is made of `DataTable.Row`, and each one of these rows is an array of cells. So let's assume that we have the following lines of code in Gherkin:

beverage	quantity
Expresso	1
Mocaccino	2

Then the resulting DataTable should be built as follows:

```
var DataTable = require('cucumber').Ast.DataTable;
var items = DataTable(); // Create empty data table
// The first row is the header of the table
items.attachRow(DataTable.Row(['beverage', 'quantity']));
// Then we add the rows with the actual data
items.attachRow(DataTable.Row(['Expresso', '1']));
items.attachRow(DataTable.Row(['Mocaccino', '2']));
```

The `dataTable` method transforms a normal array of objects into a set of rows, just like the format we used earlier. For this, it simply extracts the keys of the object to create the header row and then extracts the values for those keys for each object.

Now we can have not only an expressive Gherkin, but also the ability to invoke a step handler from another and create an alias between steps. This is a good tradeoff!

Advanced setup

In our current Gherkin, there is a lot of implicit setup. For example, in the steps that set up the order contents, we also executed code to set up the fact that the update call to the messages' DAO will not fail and that the order has no pending messages. This looks wrong; after all, we should only do the code setup needed to populate the items in the right order, which is what is stated in Gherkin.

We can solve this problem by adding additional steps:

```
Scenario: Order is empty
    Given that the order is empty
    And that the order does not have pending messages
    And that the update operations will not fail
    When the customer displays the order
    Then no order items will be shown
    And "0" will be shown as total price
    And there will only be possible to add a beverage
```

```
Scenario: Non empty order
    Given that the order contains:
        | beverage | quantity |
        | Expresso | 1           |
        | Mocaccino | 2           |
```

And that the order does not have pending messages

And that the update operations will not fail

When the customer displays the order

Then the following order items are shown:

beverage	quantity
Expresso	1
Moccaccino	2

And "6.10" will be shown as total price

And there will be possible to:

action	for item
place order	
append item	
edit item quantity	1
remove item	1
edit item quantity	2
remove item	2

We also need to add the step handlers:

```
this.Given('that the order does not have pending messages',  
sugar(function () {  
    this.messages = this.messageStorage.alreadyContains({  
        id: this.order.id,  
        data: []  
    });  
});  
  
this.Given('that the update operations will not fail', sugar(function  
() {  
    this.messageStorage.updateWillNotFail();  
}));  
  
var theOrderContains = sugar(function (orderItemExamples) {  
    this.order = this.orderStorage  
        .alreadyContains(order.withItems(orderItemExamples));  
});
```

Now the step handlers do only what is stated in Gherkin. However, we now have new problems:

- We have a duplication of setup between different scenarios.
- The setup is a bit verbose.

- There is some coupling between the "order has no pending messages" and the "order contains" steps. The first step must always be run after the second one, because it needs the order identifier. This is not good because someone can reorder the steps without being aware of this technical and business-level irrelevant detail.
- It follows from the last point that it can be argued that whether the updates fail or not is a lower level of abstraction, and this should not be mentioned in Gherkin.

Before solving these problems, let's make matters worse. If you show the nonempty order to somebody, they will frown and ask, "Why is the order price 6.10, what is the price of each beverage?" In the last chapter, we decided not to include the unit price in our examples and to encapsulate those details in the beverage example factory. After all, Mocha is for developers, and any developer can browse to the example factory to see any detail they need. However, the point of using Gherkin is to be able to communicate with nonengineers, such as domain experts, and other stakeholders, so we should include explicitly in Gherkin any information needed to make sense of the scenario. Since we cannot calculate the total price of the order without knowing the unit price of each beverage, we will need to be explicit about this:

```
Scenario: Order is empty
  Given that the shop serves the following beverages:
    | beverage | price |
    | Espresso | 1.50 |
    | Mocaccino | 2.30 |
  And that the order is empty
  And that the order does not have pending messages
  And that the update operations will not fail
  # Skipped for brevity

Scenario: Non empty order
  Given that the shop serves the following beverages:
    | beverage | price |
    | Espresso | 1.50 |
    | Mocaccino | 2.30 |
  And that the order contains:
    | beverage | quantity |
    | Espresso | 1 |
    | Mocaccino | 2 |
  And that the order does not have pending messages
  And that the update operations will not fail
  # Skipped for brevity
```

Gherkin-driven example factory

This is not a trivial change. Previously, we have been working with a fixed set of beverage examples, and now this set is defined by Gherkin. Since we do not need `test/support/examples/beverages.js` anymore, we should remove it. We then need to change the `test/support/examples/orders.js` file:

```
var counter = 0;

// Skipped for brevity
module.exports = {
    // Skipped for brevity
    withItems: function (items) {
        counter++;
        return {
            id: "<non empty order " + counter + ">",
            data: items
        };
    },
    // Skipped for brevity
};
```

We have removed the `test/support/examples/beverages.js` module and all the code that depends on it. Where should we put this code? The answer is to put it in the same module that is going to implement the new logic about beverage examples. A good place where we can put any logic relating to Gherkin examples is in our custom world object:

```
// Skipped for brevity
function beverageExampleFactoryFor(beverageExamples) {
    var factory = {};
    beverageExamples.hashes().forEach(function (beverageExample) {
        var name = beverageExample.beverage,
            factoryName = name.toLowerCase(),
            price = Number(beverageExample.price);
        factory[factoryName] = function () {
            return {
                id: factoryName + " id",
                name: name,
                price: price
            };
        };
    });
}
```

```
    return factory;
}

module.exports = function (cb) {
  var world = {};
// Skipped for brevity
  function asOrderItem(itemExample) {
    return {
      beverage: world.beverageFactory[itemExample.beverage.
toLowerCase()](),
      quantity: Number(itemExample.quantity)
    };
  }

  world.shopServesBeverages = function (beveragesExamples) {
    world.beverageFactory = beverageExampleFactoryFor(beveragesExampl
es);
  };

  world.asItems = function (itemExamples) {
    return itemExamples.hashes().map(asOrderItem);
  };

// Skipped for brevity
  cb(world); // We are done!
};
```

Our world object now has `shopServesBeverages` that will be able to dynamically construct a beverage example factory object based on the provided examples. This factory has exactly the same convention as the static one we had in `test/support/examples/beverages.js`. The actual logic is in the `beverageExampleFactoryFor` private method. It simply converts the Cucumber data table to an array and iterates it to generate a new factory function for the corresponding beverage. All of these functions are attached to the resulting example factory object.

In addition, we added an `asItems` method that contains the logic that we had earlier in `test/support/examples/orders.js` to create example order items.

We made a lot of changes in our test support code, so we need to update our existing step handlers and write a new handler for the step about "shop serving beverages":

```
var theOrderContains = sugar(function (orderItemExamples) {
  this.order = this.orderStorage.alreadyContains(
    order.withItems(this.asItems(orderItemExamples)))
});
```

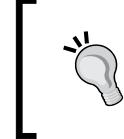
```

}) ;
var theFollowingItemsAreShown = sugar(function (orderItemExamples) {
    return expect(this.result).to.eventually
        .have.property('items')
        .that.is.deep.equal(this.asItems(orderItemExamples));
});
this.Given("that the shop serves the following beverages:",
sugar(function (beveragesExamples) {
    this.shopServesBeverages(beveragesExamples);
}));
```

Implicit versus explicit setup

In the last section, we saw that we need to explicitly add any information needed to understand the scenario outcome, looking only at the Gherkin code. This forced us to add the price explicitly. What about the "empty messages" and the "updates will not fail" steps?

By default, it is logical to assume that an order will not contain pending messages; after all, nobody will expect to see any message unless something wrong has happened. The same thing applies to the update DAO operation failing. It is even worse in this case.



If you have any doubt about what can be safely assumed by default and which information is relevant and which is not, there is a simple solution: consult with the domain expert and/or the relevant stakeholders!

In these cases, it is simply better to remove this kind of thing from the Gherkin code. After removing the steps, we can also remove the corresponding step handlers. However, we still need to execute the code that was in those step handlers so that the test will run correctly!

We can move the code about "updates not failing" to the test double itself in the `test/support/storageDouble.js` file:

```
module.exports = function () {
    // Skipped for brevity
    storage.updateWillNotFail = function () {
        dao.update.callsArgWithAsync(1, null);
    };
}
```

```
storage.toExpectUpdate = function (entity) {
  expect(dao.update).to.be.calledWith(entity);
};

// Default behavior
storage.updateWillNotFail();

return storage;
};
```

The code that states that, by default, the current order has no pending messages is more tricky, since it cannot be executed before creating the order. We can encapsulate this logic in the world object:

```
'use strict';

var newStorage = require('./storageDouble'),
  orderSystemWith = require('../lib/orders'),
  DataTable = require('cucumber').Ast.DataTable,
  Row = DataTable.Row,
  order = require('./examples/orders');

// Skipped for brevity
module.exports = function (cb) {
  var world = {};
  orderStorage = newStorage(),
  messageStorage = newStorage();

  world.orderSystem = orderSystemWith({
    order: orderStorage.dao(),
    message: messageStorage.dao()
  });

  // Skipped for brevity
  world.currentOrderHasItems = function (itemExamples) {
    world.order = orderStorage.alreadyContains(
      order.withItems(world.asItems(itemExamples))
    );
    // By default, and order has no pending messages
    world.orderMessages = messageStorage.alreadyContains({
      id: world.order.id,
      data: []
    });
  }
};
```

```
        });
    };

    world.asActions = function (actionExamples) {
      return order.actionsForOrderFrom(world.order, actionExamples);
    };
    // Skipped for brevity
    cb(world); // We are done!
  };
}
```

Now the world object imports the order example factory and has two new methods: **asActions** and **currentOrderHasItems**. The first method simply transforms a set of Cucumber action examples into actions belonging to the current order. The second one contains the logic about dealing with the storage double objects to create an order with specific contents. As discussed, the created order will not contain any messages by default.

Since all of our logic about messages and orders is inside the world object, we can make the storage doubles private. The step handler code changed to the new world API is as follows:

```
'use strict';

var chai = require('chai'),
  expect = chai.expect,
  sugar = require('../support/cucumber_sugar');

chai.use(require("sinon-chai"));
chai.use(require("chai-as-promised"));

module.exports = function () {
  // Skipped for brevity
  var theOrderContains = sugar(function (orderItemExamples) {
    this.currentOrderHasItems(orderItemExamples);
  });
  // Skipped for brevity
  var thereWillBePossibleTo = sugar(function (actionExamples) {
    var expectedActions = this.asActions(actionExamples);

    return expect(this.result).to.eventually
      .have.property('actions')
      .that.have.length(expectedActions.length)
      .and.that.deep.include.members(expectedActions);
  });
  // Skipped for brevity
};
```

The importing of the order example factory has been removed. The step handler implementations are now much simpler because most of the logic is now in the world object.



As a rule of thumb, you should use the world object to encapsulate the setup logic and the transformation between Cucumber examples and data that your application needs.



The Background section

Finally, we can remove the duplication using a **Background:** section in our Gherkin:

```
Feature: Customer displays order

Background:
  Given that the shop serves the following beverages:
    | beverage | price |
    | Espresso | 1.50 |
    | Mocaccino | 2.30 |

  Scenario: Order is empty
    Given that the order is empty
    When the customer displays the order
    # Skipped for brevity

  Scenario: Non empty order
    Given that the order contains:
      | beverage | quantity |
      | Espresso | 1           |
      | Mocaccino | 2           |
    When the customer displays the order
    # Skipped for brevity
```

The steps in the **Background:** section will be run before each scenario, so we can refactor the common steps of all scenarios relating to the same feature into this section.



If you are thinking of refactoring the common steps between features, I am sorry to disappoint you; this is not possible in Cucumber.



Parameterized scenarios

As with Mocha, sometimes we would like to triangulate in Gherkin; the equivalent would be to execute the same scenario but with different data. To do so, Gherkin provides the `Scenario Outline:` and `Examples:` sections.

Suppose you have the following scenario for the validator of the *Chapter 2, Automating Tests with Mocha, Chai, and Sinon*:

```
Scenario: Invalid number
  Given the default set of validation rules
  When the system validates the number "-2"
  Then the result will include the error "non positive"
```

We can parameterize as follows:

```
Scenario Outline: Invalid number
  Given the default set of validation rules
  When the system validates the number "<input number>"
  Then the result will include the error "<expected error>"
```

`Examples:`

input number	expected error
0	non positive
-2	non positive
3	three
5	five
15	three
15	five

In the example, we can see the `<input number>` and `<expected errors>` placeholders. A placeholder defines a variable that will be substituted by Cucumber.js for the corresponding value in the examples table. Placeholders are defined using the `<` and `>` characters enclosing their names. As you see in the example, the name can contain spaces.

Cucumber.js will execute `Scenario Outline:` once for each data row in the `Examples:` section. In each execution, it will first replace any placeholder with the corresponding value, and then it will try to find the corresponding step handler to execute.



One Scenario Outline: section can have more than one Examples: section.



Unfortunately, we cannot have nested tables in Gherkin. A nested table would make our Gherkin very complex to read, and this will remove a lot of value from the Cucumber approach.

We can now parameterize our "nonempty order" scenario:

Background:

Given that the shop serves the following beverages:

beverage	price
Expresso	1.50
Mocaccino	2.30
Frapuccino	4

// Skipped for brevity

Scenario Outline: Non empty order

Given that the order contains:

beverage	quantity
<beverage 1>	<quantity 1>
<beverage 2>	<quantity 2>

When the customer displays the order

Then the following order items are shown:

beverage	quantity
<beverage 1>	<quantity 1>
<beverage 2>	<quantity 2>

And "<expected price>" will be shown as total price

And there will be possible to:

action	for item
place order	
append item	
edit item quantity	1
remove item	1
edit item quantity	2
remove item	2

Examples:

beverage 1	quantity 1	beverage 2	quantity 2	expected price
Expresso	1	Mocaccino	2	6.1
Mocaccino	2	Expresso	1	6.1
Frapuccino	5	Mocaccino	3	26.9

Note that we added a new beverage, the Frapuccino, and replaced the data table cells and the total price with placeholders.



In fact, we can put placeholders in any part of the step, including data cells or even headers in the data tables.



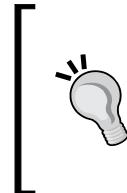
The lack of nested tables forces us to make the Examples: section a bit more verbose, using several columns to fill the cells of the full table. The main limitation of this is that we need to stick to a fixed number of rows in our data tables—in this case, only two items per order. The good news is that we do not need to change any code!

If you really need to triangulate using a different number of order items, you could try with the following Gherkin code:

```
Scenario Outline: Non empty order
  Given that the order contains the following "<items>"
  When the customer displays the order
  Then "<items>" will be shown as the order's content
  And "<expected price>" will be shown as total price
  And there will be possible to:
    | action           | for items   |
    | place order     |             |
    | append item     |             |
    | edit item quantity | <for items> |
    | remove item     | <for items> |

  Examples:
    | items           | for items | expected price |
    | 1 Espresso, 2 Mocaccino | 1,2       | 6.10          |
    | 2 Mocaccino, 1 Espresso | 1,2       | 6.10          |
    | 2 Frapuccino,1 Mocaccino,1 Espresso | 1,2,3 | 11.80        |
```

Instead of using tables for the order items, we used a comma-separated list of items. Each element in this list is a quantity followed by the name of the beverage. This is a very natural way of expressing example data in our scenario. We used the same approach for the actions; now, each action example has a for items column where we can put a comma-separated list of item indexes.



The data structure used in our test's code will often not have a direct correspondence with the way we structure the examples in Gherkin. Use the test support code to decouple the structure of the Gherkin examples and the structure we need in the actual models and DAOs.



The resulting steps are very expressive and the examples table is easy to read, so this is a good approach. The only problem is that the code of the step handlers needs to be changed to this new approach. First, in our world object we need to change the way we parse the item examples:

```
function asOrderItem(itemExample) {
  var regex = /\s*(\d+)\s*([^\s]+)\s*$/;
  var matches = regex.exec(itemExample.toLowerCase());
  if (!matches)
    throw new Error('<[' + itemExample + ']> is not an order item');
  return {
    beverage: world.beverageFactory[matches[2]](),
    quantity: Number(matches[1])
  };
}

world.asItems = function (itemExamples) {
  if (!itemExamples)
    return [];
  return itemExamples.split(',').map(asOrderItem);
};
```

First, the `asItems` method expects a simple string that contains the comma-separated list of items, so it no longer uses `hashes()`; instead, it employs `split(',')`. Second, `asOrderItem` does not receive an object, but a string, so we need to use a regular expression to parse it.

Now, we can change the steps:

```
this.Given('that the order contains the following "$items"',  
theOrderContains);  
  
this.Then('"$items" will be shown as the order\'s content',  
theFollowingItemsAreShown);  
  
this.Given('that the order is empty', function (cb) {  
  theOrderContains.call(this, "", cb);  
});  
  
this.Then('no order items will be shown', function (cb) {  
  theFollowingItemsAreShown.call(this, "", cb);  
});
```

The only changes are the wordings of the steps and the fact that we do use a plain string instead of a data table.

We now need to change the logic to parse the actions. This is in `test/support/examples/orders.js`:

```
actionsForOrderFrom: function (order, actionExamples) {
  var actionFactory = actionFactoryFor(order);

  return actionExamples.hashes().map(function (actionExample) {
    var actionName = toCamelCase(actionExample.action),
        forItems = actionExample['for items'];

    if (!forItems)
      return actionFactory[actionName]();

    return forItems.split(/\s*,\s*/).map(function (itemIndex) {
      return actionFactory[actionName](itemIndex - 1);
    });
  }).reduce(function (a, b) {
    // Flatten array
    return a.concat(b);
  }, []);
}
```

Now we need to map the value of the `for items` column to an array of actions. Since we can end up with an array of actions, we need to flatten the result.

We are investing a lot of effort in translating examples into actual data structures that we can use. This effort will allow us to write a Gherkin that is easy to understand, but we need to be careful here and write this kind of code in a way we can reuse it. For this, we do not just need to create reusable utilities, we also need to define both our data examples and steps in a homogeneous way, so that we can take advantage of our test support code.



Avoid writing Gherkin that looks technical! The point of writing Gherkin is to engage the stakeholders in their own language.

Finishing the feature

We can now add the scenario about the pending messages:

```
Scenario Outline: Order has pending messages
  Given that the order has the following pending messages "<pending>"
  When the customer displays the order
  Then "<pending>" messages will be shown
  And there will be no more pending messages
```

Examples:

pending	
bad quantity '-1'	
beverage does not exist, bad quantity '-1'	

Now, we need to edit test/support/example/errors.js to be able to parse the messages:

```
var errorExampleFactory = {
  asMessage: function (messageExample) {
    var regex = /\s*([^\n]+)([^\n]+)?\s*$/;
    var matches = regex.exec(messageExample);
    if (!matches)
      throw new Error('<[' + itemExample + ']> is not a message');
    var factoryName = matches[1].replace(/\s+$/, ''),
        factory = errorExampleFactory[factoryName];
    if (typeof factory !== 'function')
      throw new Error('<[' + factoryName + ']> is an unknown
message');

    return factory(matches[2]);
  },
  'bad quantity': function (params) {
    return {
      key: "error.quantity",
      params: params
    };
  },
  'beverage does not exist': function () {
    return {
      key: "error.beverage.notExists"
    };
  }
};

module.exports = errorExampleFactory;
```

We can change the `world` object as follows:

```
world.currentOrderHasItems = function (itemExamples) {
    // By default, no messages
    world.alreadyExistsAnOrderWith(itemExamples, "");
};

world.currentOrderHasMessages = function (messagesExamples) {
    // We do not care about the items here!
    world.alreadyExistsAnOrderWith("", messagesExamples);
};

world.alreadyExistsAnOrderWith = function (itemExamples,
messagesExamples) {
    world.order = orderStorage.alreadyContains(
        order.withItems(world.asItems(itemExamples)))
);
// By default, and order has no pending messages
world.orderMessages = messageStorage.alreadyContains({
    id: world.order.id,
    data: world.asMessages(messagesExamples)
});
};

world.asMessages = function (messagesExamples) {
    if (!messagesExamples)
        return [];
    return messagesExamples.split(',').map(error.asMessage);
};
```

The new `asMessages` method will transform the message examples into an array of error message objects using the `test/support/example/errors.js` utility.

With this new scenario, it is clear that we can potentially create an order with any combination of items and messages, so we need a method for this. This method is `alreadyExistsAnOrderWith`. The `currentOrderHasItems` and `currentOrderHasMessages` methods rely on `alreadyExistsAnOrderWith`. This method is now in charge of setting up the doubles to create an order with a specified set of items and messages.

Finally, we will add the step handlers:

```
var orderHasPendingMessages = sugar(function (messages) {
  this.currentOrderHasMessages(messages);
});

this.Given('that the order has the following pending messages
"$messages"', orderHasPendingMessages);

this.Then('$messages messages will be shown', sugar(function
(messages) {
  return expect(this.result).to.eventually
    .have.property('messages')
    .that.is.deep.equal(this.asMessages(messages));
}));

this.Then('there will be no more pending messages', function (cb) {
  orderHasPendingMessages.call(this, "", cb);
});
```

This finishes the "happy path" of the feature. Now, we should start adding scenarios about what happens when one of the DAOs fails or if the requested order simply does not exist, but I will leave it as an exercise for you.

Useful Cucumber.js features

There are some features in Cucumber.js that can be handy in certain situations. Let's see what they are.

Tagging features and scenarios

You can tag features, scenarios, and Examples: sections to selectively execute only tagged parts of your specification.

You can create a tag using the @ character as follows:

```
@ready
Feature: Some feature

@simple
Scenario: scenario 1
  # Skipped for brevity
@errorcase
Scenario: scenario 2
  # Skipped for brevity
```

```

@important @regression
Scenario Outline: scenario 3
  # Skipped for brevity

Examples:
| placeholder 1 | placeholder 2 |
| example 1.1   | example 1.2   |
| example 2.1   | example 2.2   |

@complicated
Examples:
| placeholder 1 | placeholder 2 |
| example 3     | example 2     |

```

You can simply put a space-separated list of tags in any section of the Gherkin code. If you add tags at the feature level, all its scenarios will inherit it.

To run the scenarios selectively, you can use the `--tags` option:

```

$> cucumber.js --format pretty --tags @ready
$> cucumber.js --format pretty --tags @simple
$> cucumber.js --format pretty --tags @simple,@errorcase
$> cucumber.js --format pretty --tags @ready --tags @~complicated

```

The first command line will run all the scenarios that are tagged as `@ready`. In this example, this means all the scenarios in our feature, but potentially not in other features. The second command will run only scenario 1.

The third command line will run scenarios that are either tagged as `@simple` or as `@errorcase`. When a comma-separated list of tags is specified in the command line, they are combined using OR logic.

You can have several `--tags` arguments—for example, in the preceding command line. In this case, Cucumber.js will combine the expressions using AND logic.



You can use the predefined `@ignore` tag to momentarily avoid the execution of certain scenarios.

Tags can be very useful to organize your test runner. For example, one can use tags to group together features that belong to the same subsystem or to relate some of them to specific bug tickets.

Hooks

In Cucumber.js, you can define hooks—that is, functions that can be run before, after, or around each scenario.

The function that you use as a hook behaves like a normal step handler: the `this` keyword points to the world object, and you must call the callback to notify that you are done. That is why we can use our `sugar` utility for hooks too.

The before hook

You can declare a before hook using the `this.Before` function provided by Cucumber.js. For example, you could have a file called `test/step_definitions/hooks.js`:

```
module.exports = function() {
  this.Before(sugar(function() {
    // Note: "this" here points to the world object
    return this
      .writeOrdersToDB()
      .then(this.initServer.bind(this));
  }));
};
```

We used our `sugar()` utility. In the example, `writeOrdersToDB` and `initServer` return promises.

Before hooks will be executed once before every scenario, but after the world constructor. The `Background:` section, if any, will be executed after the hook.

Before hooks are useful for initialization purposes, such as setting up the DAOs or starting a server. You need to decide what part of this initialization logic should go inside the world constructor and what should be done inside a before hook. Usually, it depends on your personal coding style; some people use only the before hook, and some others do not. I often use hooks for technical set up or clean up that is not strictly relevant for the stakeholders, and hence should not appear in the Gherkin.

The after hook

We can write a symmetric after hook for the before hook we just wrote:

```
module.exports = function() {
  // Skipped for brevity
  this.After(sugar(function() {
    // Note: "this" here points to the world object
    return this
      .stopServer()
      .then(this.deleteOrdersFromDB.bind(this));
  }));
};
```

After hooks will be executed once whenever a scenario finishes.

The around hook

Around hooks are a bit more complex. They allow us to execute code before and after each scenario, giving us more fine-grained control. For example, we can merge both the before and after hooks we just wrote as follows:

```
module.exports = function() {
  this.Around(function(executeScenario) {
    // Note: "this" here points to the world object
    this.writeOrdersToDB()
      .then(this.initServer.bind(this))
      .then(function() {
        executeScenario(function(cb) {
          this.stopServer()
            .then(this.deleteOrdersFromDB.bind(this))
            .then(cb, cb);
        });
      });
  });
};
```

The around hook receives an `executeScenario` function. We should call this function whenever the scenario must be executed. The function receives a callback as a parameter with the code block (the same code as in the after hook) to be executed after the scenario is run.

Non-English Gherkin

What happens if our stakeholders do not understand English? We can simply write the Gherkin code in a language they are comfortable with!

The keywords of Gherkin, such as `Scenario:` or `Feature:`, are translated to other languages (at the moment of writing this book, they are translated into 40 languages!).

Just add `# language: code` as the first line of your feature file. The code is the two-letter code of the language you choose (`en` for English, `es` for Spanish, and so on). To see what languages are supported and to find out the translations of each keyword, visit <https://github.com/cucumber/gherkin/blob/master/lib/gherkin/i18n.json>.

Cucumber.js or Mocha?

Which is better? Cucumber.js or Mocha? There is no clear answer to this question, but here is my advice: if you have engaged stakeholders who are willing to read or even write the Gherkin code, go for Cucumber.js. If you are not in this (fortunate) situation, then using Cucumber.js is not so attractive.

Even if you are in this situation, you need to think twice before using Cucumber.js, specially if you are new to BDD or JS. The main problem with Cucumber.js is that you need to invest a lot in the test support code that translates the Gherkin to real code. This effort is wasted if the stakeholders are not going to, at least, read the Gherkin code and give you feedback on the intended behavior of the system.

On the technical side, the Cucumber.js tool itself is a bit immature, specially compared to Mocha. I specially miss the support for promises, better assertion failure reporting, a simple API to deal with tables, or a simple way to call one step from another. These are problems that could be solved in the future or with a couple of utilities, but nowadays you will pay an extra cost to build this small helper utilities.



The Ruby version of Cucumber is much more mature, and it is easier to work with it.



However, even when I am writing Mocha, I start thinking in the tests from the Gherkin point of view. I can recommend that you think of your features and scenarios as if you were going to write them with Gherkin and then adapt them to Mocha. If you have a look at the previous chapter, the structure and titles of the test suite are very much influenced by how I would have done it with Gherkin.

Another option is to use a tool that transforms Gherkin to Mocha. Definitely, this is possible, as the project at <https://github.com/mklabs/mocha-gherkin> shows.

Summary

In this chapter, you learned how to use Gherkin and Cucumber.js to write BDD features. This has been a long and hard chapter, because you not only needed to learn a new tool, Cucumber.js, but you also needed to build a set of utilities to make your life easier.

Always write features and scenarios in a language that makes sense and is understandable to the stakeholders. Avoid technical details, such as database structure, identifiers, or low-level operations, in your Gherkin.

It is a best practice to create a custom world object. The world object will be instantiated once before each scenario, so it is a good place to store any information that needs to be accessed by all the steps in the same scenario. In general, keep the code in the step handlers as small as possible. To achieve this, try to move some logic to the world object.

Create small utilities to allow Cucumber.js to deal with promises, synchronous steps, and Chai assertions more easily.

In Gherkin, you can create data tables to model complex examples that contain lists of objects, so it is a good practice to create test utilities that help you transform these examples and data tables into data structures that your system needs. Add entry points to this API in the world object; this way, you can take advantage of the existing context in the world object to simplify the API.

Another good trick is to share the step handler functions across different step definitions. This is useful for the implementation of steps that are simply aliases, or slight variations, of each other.

Avoid duplication in the setup of your scenarios that refactor common setup steps to a `Background:` section.

You can use `Scenario Outline:` to create a parameterized scenario. The scenario will be run once for each row in the `Examples:` section.

Until now, you have learned how to test your business logic first, but sometimes we would like to test other aspects of our application, such as the UI or how we are publishing our logic to the Web. In the next chapter, we will start exploring how to test the server side of our application.

5

Testing a REST Web API

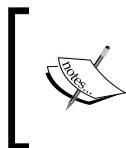
In this chapter, you will learn how to test a REST web API. As we will see, publishing a specific functionality as a REST web API is not simple and involves a lot of corner cases. Fortunately, there are a lot of tricks we can use to make our life easier.

In this chapter, you will learn the following topics:

- How to make a very fast test suite that can run around 150 tests per second to drive a REST API
- Why we should isolate the tests of the REST API from the tests of our business layer and how to do this efficiently
- The main design patterns for our test suite when dealing with REST APIs

The approach

We have already coded a feature for the myCafé application; now, we have been asked to publish this feature as a web API. Business thinks that other parties can build nice applications on top of it, and IT thinks that we can integrate the order subsystem as a microservice inside the company.



If you are new to restful web API's design, you can get an introduction to it in RESTful Java Patterns and Best Practices, available at <https://www.packtpub.com/application-development/restful-java-patterns-and-best-practices>.

As an industry, it is a good practice to publish our logic as a REST service that follows the hypermedia approach. For this, we can use a standard mime type called `application/hal+json`; this is simply known as **HAL**.



If you want to know the exact details of HAL, you could read the standard documentation at <http://tools.ietf.org/html/draft-kelly-json-hal-06>.

Basically, we need to create a web server that receives a GET HTTP request for the order, calls the `display()` method of the core logic we have developed so far, and maps the response to a HAL document. It does not sound very complex, but how do we test it?

A strategy to test web APIs

A naïve approach to testing would be to simply change the existing tests to attack a web server instead of a local API. We can maintain the current features and scenarios and change the step handlers, if we use Cucumber.js, or the implementation of the tests, if we use Mocha.

This approach is called an **integrated test**, since we are testing the core business layer (the order subsystem) and the web layer at the same time. This is actually a popular approach and can work pretty well with simple cases such as the one we are using in this book. However, it will not scale very well for non-toy projects.

The problem here is that creating a hypermedia web API for the order subsystem, and the order subsystem itself, are different problem domains.

The main point is that the web API does not really care about business logic; it simply deals with the technical aspects of REST and HAL. It could even publish as a HAL document, an order object that is not strictly valid according to the business logic.

On the other hand, the order subsystem must be totally independent of how we publish and consume it. It can be consumed as a local API, published as HAL, as an old SOAP style service, or through an enterprise queue system.

The following are the main responsibilities of the web API:

- Publishing the `display()` operation in HTTP. This involves defining an URI, which HTTP method will be used, which HTTP status codes to use, and so on.
- Extracting the corresponding parameters from such a request.
- Mapping the result of the `display()` operation as a HAL document. This involves technical details, such as how to model the actions or how to represent the order items.
- Dealing with network problems.

To sum up, the main responsibility of the web API layer must be to publish the business processes over the web. A corollary of this is that the behavior of the web API layer should not add any kind of extra logic to the business rules. Also, for each business operation, there must be a feature in the web API that publishes it over the Web.

As we just saw, the responsibilities of the web API layer are mostly independent of the business logic ones, and they are complex enough to be considered a different problem domain.

If we decide to make an integrated test, we might face some of the following drawbacks:

- If we want to have good test coverage of all the paths, then we will end up with a combinatorial explosion of scenarios and features. This happens because we need to test for most combinations of the functionality of both layers.
- The tests are confusing and usually not at the right level of abstraction. If we test only at the business level, then we will have a big gap between the description of the test and the actual code we need to implement in the test, thus leading to complex tests. Furthermore, since the test is defined in the language of the business, we cannot precisely specify tests for technical aspects of the web API (the tests may be getting complex due to this big gap in abstraction). On the other hand, if we test at the web API level, the resulting tests will not be very user-friendly, and we might get lost in technical details instead of focusing on functionality.
- An integrated test is hard to diagnose. If a test fails, where is the problem? In the web layer or in the business layer?

Of course, in small projects, these factors are not very big, and we can go for an integrated test approach and test both layers together. However, I have seen too many ineffective, over-complex, and expensive-to-maintain test codebases due to this mistake. So, it is better to slice the system into two dimensions. One axis is the different layers of abstraction that we can have: the web API layer and the business layer. The other one is based around business entities: orders, beverages, and so on. We will have a separate library, each one with its own separate feature set with a corresponding test suite, stated in a language that corresponds with its own problem domain.



If you are a domain-driven design practitioner, note that the business entities I proposed correspond to aggregate roots or bounded contexts.

Mocha or Cucumber.js?

Which tool should we use? In this case, the problem domain is very technical, and we will normally have different stakeholders from the ones we have for the business logic layer. For the later ones, it is enough to know that the order is published as a "service"!

The point is that any stakeholder who can understand the problem domain of publishing a web API can probably read the code of your tests, provided that they are maintainable enough.



The conclusion is that Cucumber.js will not, probably, add much value to the tests of this technical problem domain, so it is better to use Mocha.



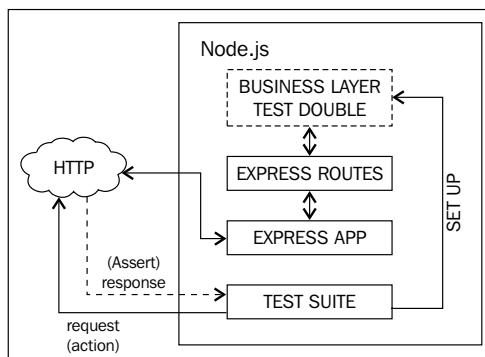
The plan

Both the order subsystem and its web API are different problem domains, so we should test the latter as a different set of features of our application.

For this, we will make a test double of the business layer; in this case, it is only an object with a `display()` method. We need to design a server API that we can use to start, stop, and make the server use the test double.

We can use any HTTP client for Node.js to drive the interaction with the server. The other option could be to use a mocking library for the network stack, but I consider this kind of mocking only necessary if you really need more speed in your tests. However, even using real HTTP calls, we can run around 100 tests per second on a normal laptop. This is usually sufficient to test-drive our server-side code.

Hence, the proposed testing architecture is like the following diagram:



Testing architecture for our web API

As you can see, both the server and test code reside in the very same Node.js process. Before running any test, the server will be started and will not be stopped until the end of the test suite. Before each test, the server is injected with a brand new test double that represents the order core business layer, so we can run each test using a clean and predictable state.

All this arrangement makes our test suite faster due to the following reasons:

- No need to use separate processes for the server and the tests.
- No need to instantiate, bind to a server socket, and stop our server app for each test. This saves a lot of time, as these steps are very expensive.
- No need to set up a database. After all, we are testing only the web layer so that we can perform the setup phase with only a simple test double in memory.

If you need an even faster test, you can opt to use a library to make test doubles of the Node.js HTTP server libraries or, if you are using it, the `express` package. In this case, you can apply the techniques we explored in the previous chapters.

Testing the GET order feature

Let's create a brand new package in a different project. Why a different project? We have explored the idea that the API deserves its own set of features and that we should decouple it from the core layer using a well-defined interface that we will mock. Since it is a different problem domain, we should create a different package. This way, we can develop them and manage their life cycles in an independent fashion.



For a simple project like this, it is OK if you prefer to simply put both layers in the same package.

So, let's create a `mycafe-api/` folder with the same project layout we have been using so far. Initialize the project by creating a basic `package.json` file, and then, run the following commands to install the packages we need:

```
$ ~/mycafe-api> npm install --save-dev chai sinon sinon-chai chai-as-promised request mocha
$ ~/mycafe-api> npm install --save express q
```

Note that I am installing `express` as a runtime dependency (version 4.x). This is a great package that allows us to create either a web application or a web API. Due to brevity, we will not see how to use it in this book; in fact, it deserves its own book. For more details, just see <http://expressjs.com/4x/api.html>.

The other dependencies are the normal testing libraries we have been using so far. As I explained earlier, we will use Mocha instead of Cucumber.js.

However, how do we connect this web API package with the business layer? In a real system, the project we did in the last two chapters will be the core business logic layer. We can publish this project as an NPM package to our private NPM repository or link it as a local package on our machine. Then, we can connect the web API package and the business logic package in the following two ways:

- One way is by referencing the business logic package as a runtime dependency from the `mycafe-api` project.
- An even better alternative would be to have a `mycafe-server` package, with runtime dependencies to the business logic package, and the web API package. This server package can instantiate a business logic instance, inject it to the web API package, and start the web server using the routes from the web API package.

Now we are ready to implement our first scenario.

Exploring our feature a bit

What should be our first feature? In a web API, the equivalent of displaying an order is to retrieve the web resource that corresponds to that order. The standard way to do so involves issuing a GET HTTP request to the order resource that should be published in a URI. Hence, our feature should be called **Get Order** or something along these lines.

Basically, we need to explore what happens when we issue a GET request against our system. There are several scenarios here: success, the requested order does not exist, security, an error on the database, and so on.

Let's create a `test/get_order.js` file with the following contents:

```
'use strict';

var chai = require('chai'),
    expect = chai.expect;
```

```
chai.use(require("sinon-chai"));
chai.use(require("chai-as-promised"));

describe('GET /order/:orderId', function () {
  context('The order exists', function () {
    it('will respond with a 200 code');

    describe('will respond with a HAL document for the order',
      function () {
        // What goes here?? We will see throughout this chapter
      });
  });
});

context('The order does not exists', function () {
  it('will respond with a 404 code');
});

context('The order subsystem is down', function () {
  it('will respond with a 500 code');
});
```

To start with, we can have one success and two failure scenarios:

- If the order is found, we must return a 200 Ok code. We also need to describe what kind of document we want to return in the body of the request. Later in this chapter, we will see what to do here.
- If the order requested does not exist, we should return a 404 Not Found code.
- If there is an exception or error during the execution of the order, we should return a 500 Internal Server Error code.

I am sure that, if we think hard enough, we can add more error scenarios. For brevity, we will focus on the success scenario. I will leave the other scenarios to you as an exercise. For now, let's start with the very basics: testing the 200 Ok code.

Also notice the title of the 'GET /order/:orderId' feature; it is very technical, but since our problem domain is how to create a web API, we should use titles that are meaningful at the abstraction level of a web API. In this case, this title makes sense because the actual action that we are testing is what happens when a client performs a GET request to the /order/:orderId URI, where :orderId means the actual public identifier of the order.

Starting, stopping, and setting up our server

To start testing, we need to build some support code that allows us to start and stop the server, to easily make requests to it, and to create a test double for the business layer.

Assuming that we are using express as our server toolkit, we can create a `test/index.js` file with the following lines of code:

```
'use strict';

var sinon = require('sinon'),
    express = require('express'),
    app = express(),
    server,
    port = process.env.PORT || 3000;

before(function (cb) {
  app.listen(port, function () {
    server = this;
    // Note: arguments contains the arguments passed to this function,
    // in this case a potential error produced by app.listen()
    cb.apply(this, arguments);
  });
});

after(function (cb) {
  if (!server)
    return setImmediate(cb);
  server.close(cb);
});
```

We are just using the standard express API in combination with the `before()` and `after()` Mocha calls. They will run only once before and after all the scenarios, so we will reuse the same server instance throughout all the tests. Mocha will run both the `test/index.js` and `test/get_order.js` files and sort out the correct order of execution.

However, for each test, we need a new clean setup. For this, we need to create a brand new test double for the business layer, so we can add the following lines of code:

```
beforeEach(function () {
  var orderSystemDouble = {
    display: sinon.stub()
  };
  // What we do with it???
});
```

Now we need to pass the double to the code that does the actual work. Since we are using `express`, it seems natural that the package we are testing will export a function that takes the order subsystem and returns an `express.Router()` instance that can be mounted in any server. So, we need to require such a function and use it in our setup:

```
var sinon = require('sinon'),
    newRouteFor = require('../index'),
    express = require('express'),
    app = express(),
    server,
    port = process.env.PORT || 3000;

// Skipped for brevity
beforeEach(function () {
    var orderSystemDouble = {
        display: sinon.stub()
    };
    app.use('/orders', newRouteFor.order(orderSystemDouble));
});
```

In this example, the function we are trying to implement using BDD is `newRouteFor.order(orderBusinessLogic)`. In the preceding code, we passed a test double as the order business logic instance, instead of a real one. By convention, in Node, you often expose the public API of the package in an `index.js` file in the root of the package, so we did the same here.



As we explained before, this design allows a main server package to create an instance of the real business logic, pass it to the web API package, and obtain a route. This route can be used to start the server. This way we have a modular server, and we can switch to another implementation of the business layer if necessary.

There is only one caveat, since `beforeEach()` will be invoked once before each test, we will end up using several route instances for the same path in the same `express` application. This will force `express` to call each one of the routes it has whenever it receives an order request, processing the request multiple times. We do not want that, so we need to complicate our approach a little to use only one instance of router:

```
'use strict';

var sinon = require('sinon'),
    newRouteFor = require('../index'),
```

```
express = require('express'),
app = express(),
server,
currentOrderSystem,
port = process.env.PORT || 3000;

before(function (cb) {
  this.ordersBaseURI = '/orders';
  app
    .use(this.ordersBaseURI, newRouteFor.order({
      display: function () {
        return currentOrderSystem.display.apply
          (currentOrderSystem, arguments);
      }
    }))
    .listen(port, function () {
      server = this;
      cb.apply(this, arguments);
    });
});

after(function (cb) {
  if (!server)
    return setImmediate(cb);
  server.close(cb);
});

beforeEach(function () {
  currentOrderSystem = {
    display: sinon.stub()
  };
  this.orderSystem = currentOrderSystem;
});
```

First, the `orderSystemDouble` local variable was moved to a module-scoped one called `currentOrderSystem`. Now we can configure the router in the `before()` section only once using a fake order system that will simply lazily delegate to the corresponding method of `currentOrderSystem`. Provided that your router implementation is stateless, this should do the trick!

Note that we stored both the order system double and the base URI where the orders will be published—that is, in `this.orderSystem` and `this.ordersBaseURI`, respectively. This way, we can access them in our tests to create an additional setup and perform requests to the server.

Testing whether the API responds with 200 Ok

Since we are testing the success scenario, we should add some setup in the `test/get_order.js` file:

```
var chai = require('chai'),
    expect = chai.expect,
    Q = require('q');

chai.use(require("sinon-chai"));
chai.use(require("chai-as-promised"));
describe('GET /order/:orderId', function () {
  beforeEach(function () {
    this.orderId = "<some order id>";
    this.orderURI = this.ordersBaseURI + '/' +
      encodeURIComponent(this.orderId);
  });
  context('The order exists', function () {
    beforeEach(function () {
      this.orderModel = {};
      this.orderSystem.display
        .withArgs(this.orderId)
        .returns(Q.fulfill(this.orderModel));
    });
    it('will respond with a 200 code',
      // Skipped for brevity
    );
    // Skipped for brevity
  });
});
```

There is a first `beforeEach()` block that affects the whole test suite. Here, we defined the identifier of the order we are going to use throughout our feature and store it in `this.orderId`. We also specified the exact URI we need to use for an order in the `this.orderURI` field. This is a very important detail, since it is part of the API contract. In this case, we used the `/orders/:orderId` format, with the caveat that we need to encode `orderId` to escape illegal characters that cannot be part of a URI path segment.

The next `beforeEach()` block is specific to the setup of the scenario and uses `Q` to make the `display()` method of the order system double return `this.orderModel` when asked for the correct id. In the `this.orderModel` field, we stored the example of the model that it is supposed to be returned by `display()`.

We can implement the setup of the two error scenarios in a similar way:

```
context('The order does not exists', function () {
  beforeEach(function () {
    this.orderSystem.display
      .withArgs(this.orderId)
      .returns(Q.fulfill(null));
  });
  it('will respond with a 404 code');
});

context('The order subsystem is down', function () {
  beforeEach(function () {
    this.orderSystem.display
      .withArgs(this.orderId)
      .returns(Q.reject(new Error('Expected error')));
  );
  it('will respond with a 500 code');
});
```

The setup is very similar, but we returned a `null` to simulate that the business layer was not able to find the order. In the other scenario, we returned a rejected promise to simulate that something went wrong in the business layer.

Should we use a realistic order object?

In the preceding code, we set this model to an empty object. However, we know from the previous chapters that the orders are not empty objects; they should at least have `totalPrice`, `messages`, `items`, and `actions` arrays, even if they are empty. Perhaps, we should change our code as follows:

```
this.orderModel = {
  totalPrice: 0,
  messages: [],
  items: [],
  actions: []
};
```

After all, this is the correct empty order, and the specification of the `display()` operation ensures this. Should we always work with orders that are well formed in our tests? There is no clear answer to this. If we do so, our code probably will be simpler, because the set of model instances it needs to deal with is more constrained. For example, we do not need to check whether the `actions`' field exists or not; we already know that there will always be at least one empty array.

On the other hand, if we really want the API layer and the business logic layer to evolve as independently as possible, we will need both layers to be as decoupled as possible. Not expecting a well formed order will make our API layer more robust against trivial errors or design changes in the business layer. For example, the business layer team could change their minds and, for whatever technical reason, decide that, if the order does not have messages, it will simply not have a messages field. This could break our API layer if we had relied on this trivial detail when we created our tests for the API layer. This couples both subsystems and makes evolution difficult for both of them. Avoiding knowledge about what a well formed order is, has the added advantage that the setup of our tests will be simple, as we only need to define the order's fields that are strictly relevant to our API, even if the data in those fields are not strictly correct from the point of view of the business layer.

As a conclusion, if both layers are owned by the same team, the API layer will only talk with the same implementation of the business layer, going for a more realistic order could be better. On the other hand, if you think that the risk of trivial misunderstandings between teams is high, probably specifying only the strictly relevant information in the order for our test is a good approach.

Implementing the test

We need to perform an HTTP GET request to the API in order to test whether it returns a HTTP status code 200. It would be nice to be able to have the following test implementation:

```
it('will respond with a 200 code', function () {
  return expect(this.GET(this.orderURI)).to.eventually
    .have.property('status', 200);
});
```

The idea is that the `this.GET(URI)` method will perform the HTTP request and return a promise with an object that describes the response. We can write this kind of utility using the `request` module. Let's create `test/support/client.js` to hold this utility using the following code:

```
'use strict';

var request = require('request'),
  Q = require('q');

module.exports = function (baseURL) {
  return {
    GET: function (resourcePath) {
      return Q.Promise(function (resolve, reject) {
        request({
```

```
        timeout: 500,
        uri: baseURL + resourcePath,
        method: 'GET',
        headers: {
          'Accept': 'application/hal+json'
        }
      }, function (error, response, body) {
        try {
          if (error)
            return reject(error);
          resolve({
            status: response.statusCode,
            body: body ? JSON.parse(body) : undefined
          });
        } catch (err) {
          reject(err);
        }
      });
    );
  );
};
```

We exported a factory function that takes a base URL and returns a very basic web API client. For now, this client is an object with only a `GET(resourcePath)` method. This function will use the `request` library to perform the GET HTTP request. We are also explicitly specifying that the mime type to be returned is `application/json+hal`.

To make the `request` library return a promise, we used `Q.Promise()`. This will return a promise that will be successfully fulfilled when the code calls the `resolve()` callback. It will be rejected if we call `reject()`.



We can simply use the basic HTTP client that comes with Node.js but I found the `request` module quite handy. For details on it, have a look at <https://github.com/mikeal/request>.

Now let's modify our `test/index.js` file to use this utility:

```
var sinon = require('sinon'),
newRouteFor = require('../index'),
newClient = require('./support/client'),
```

```
express = require('express'),
app = express(),
server,
currentOrderSystem,
port = process.env.PORT || 3000;

before(function () {
  this.GET = newClient('http://localhost:' + port).GET;
});

// Skipped for brevity
```

We used the `before()` block to create an instance of the web API client and then attached the `GET(resourcePath)` method only once to the runtime context of Mocha.

You can now run the test; it will fail because we have not yet written any code for the server. Since this is not a book about Express, I will leave it to you as an exercise to make the tests pass. The important point here is that the tests are fast, and you can use them to test drive your API server code.

Testing our HAL resource for orders

Apart from saying that we should return a `200 Ok` code, we should describe the document we return in the body of the response.

Since it is a HAL resource and we are building a hypermedia API, it would be nice to have a `self` link that points to the order itself. This is a best practice since it allows the client to know the URI of the resource, even if it is embedded in another one. Let's create a test for this:

```
describe('will respond with a HAL document for the order', function () {
  it('will have a self link', function () {
    return expect(this.GET(this.orderURI)).to.eventually
      .have.deep.property('body._links.self')
      .that.is.deep.equal({
        href: this.orderURI
      });
  });
});
```

We simply test whether the body of the request transports a HAL document with a `_links` property. In a HAL document, the `_links` property is an object that contains links to other resources. Each one of the keys of `_links` represents a link type; in this case, we are interested only in the `self` link, which is the standard link type that defines a link to the resource itself. The point here is that we are testing at the right level of abstraction for our web API. After all, we are concerned with the details of how we publish the order models and not with other aspects of the system. This involves testing the details of HTTP communication, such as response codes and URIs, and also testing the format of the data returned. In our example, we want to ensure that the returned document complies with the HAL standard and that it is useful for potential consumers. If we were making an integrated test using the scenarios we defined in *Chapter 3, Writing BDD Features* and *Chapter 4, Cucumber.js and Gherkin*, we would not be able to test all of this effectively, since these scenarios are defined at a business level.

We can write similar tests for the data contained in the `order`, `totalPrice`, and `messages` properties:

```
describe('will respond with a HAL document for the order', function () {
  // Skipped for brevity
  it('will have a totalPrice property with the total price of the
     order', function () {
    this.orderModel.totalPrice = 222;

    return expect(this.GET(this.orderURI)).to.eventually
      .have.deep.property('body.totalPrice', this.orderModel.
        totalPrice);
  });
  it('will have a messages property with the pending messages of the
     order', function () {
    this.orderModel.messages = ['msg1', 'msg2'];

    return expect(this.GET(this.orderURI)).to.eventually
      .have.deep.property('body.messages')
      .that.is.deep.equal(this.orderModel.messages);
  });
});
```

In this case, the HAL document should have corresponding `totalPrice` and `messages` properties, with exactly the same data as those of the order. Again, we do not care about the fact that the order has the wrong price (it should be 0, since it has no items) or what exactly a message should be (should it be a string, or should it be a more complex object?). The point is that all of these details are the responsibilities of the business layer, and it would be a waste of time to test them again, and a source of coupling between both layers. After all, we do not want to break our web API layer, if someone in the future decides that, if you want to buy two cappuccinos, the second one will be for free. These are business rules that are not the responsibility of the web layer, and we should not be concerned, during the web API's testing, with what a correct order is.

The contract with the business layer

As we saw earlier, we should not include the business rules in our contract with the business layer, since they are encapsulated by this layer. This also applies to the syntactic aspects of the order model, or schema. Obviously, there must be some contract that tells us the names of the properties and the information that they hold.

For example, let's take the `totalPrice` property. Should it be a number? Can it be a string too? Or should it always be an object with the `amount` and `currency` fields? This kind of information should be in the contract, and it will shape our exact tests. If, for example, the `totalPrice` property can be a number and a string, we should add a test that says what will happen when it is a string. Do we transform it to a number or not? Should we say something about the currency? Or do we simply return the same thing in our HAL document without processing it?



I prefer to go for a very loose contract if I can. It introduces a bit more complexity on the web API layer, but it allows the business layer to be simpler. Besides, we can evolve both layers better when they're independent of each other.

What will happen if the order model instance has no `totalPrice` or `messages` properties? If this circumstance breaks our current implementation of the web API layer, perhaps, we should add tests for this. If the contract between both layers clearly states that this will never happen, then maybe it is a waste of time to test for circumstances that should never happen.

However, using defensive coding is a sensible position if you are working in a dynamic-type language, such as JavaScript, and both layers are developed by separate teams. In JavaScript, there is no compiler to enforce that the orders will always have a predefined schema, and there can be misunderstandings between the team writing the business layer and the one writing the web API layer. In JavaScript, it is a good practice to test whether, in the event of an error or a violation of the contract, our web API layer will not fail catastrophically and will handle the problem somehow.

One approach for this is to simply try to recover the error and return some meaningful default value to the user. This is a forgiving approach and the one I used in the examples earlier. On the one hand, we have a scenario, 'The order subsystem is down', to handle what happens when a severe problem occurs. On the other hand, in the 'will respond with a 200 code' and the 'will have a self link' tests, we also check whether we send at least an empty but correct HAL response, even if the business layer returns an empty object, without any totalPrice or messages fields. However, this is not enough; we need to assert what happens with the totalPrice and messages fields of the HAL document. Should they not be present in the response, or should they assume default values? If it is the former, we must test explicitly for the nonexistence of these fields. If it is the latter, we must test whether we have fields with the default value.

We can change our tests as follows:

```
describe('will have a totalPrice property', function () {
    it('with 0 as default value', function () {
        return expect(this.GET(this.orderURI)).to.eventually
            .have.deep.property('body.totalPrice', 0);
    });
    it('with the total price of the order', function () {
        this.orderModel.totalPrice = 222;

        return expect(this.GET(this.orderURI)).to.eventually
            .have.deep.property('body.totalPrice',
                this.orderModel.totalPrice);
    });
});
describe('will have a messages property', function () {
    it('with an empty array as default value', function () {
        return expect(this.GET(this.orderURI)).to.eventually
            .have.deep.property('body.messages')
            .that.is.an('array').empty;
    });
});
```

```
it('with the pending messages of the order', function () {
  this.orderModel.messages = ['msg1', 'msg2'];

  return expect(this.GET(this.orderURI)).to.eventually
    .have.deep.property('body.messages')
    .that.is.deep.equal(this.orderModel.messages);
});

});
```

These tests now clearly state that, if there is no `totalPrice` property in the order, we assume `0` as the default value. For messages, we assume an empty array. This is a very nice approach, since it makes the life of the web API consumer easier because it does not need to check whether these properties are `undefined` or not. It has the slight disadvantage that the HAL document is a bit bigger and wastes more bandwidth. As always, these kinds of decisions involve a trade-off.

If we could not decide on a good default value, maybe because there are business rules involved in this, then the best option would be to simply test whether the fields are included or not in the message.

Note that the preceding approach would lead to exactly the same tests as if we had defined a loose contract where the order model object does not need to have a `totalPrice` or `messages` field. Because of that, it is better to define the contract like this and simplify the contract with the business layer at the syntactic level.

If we decide that we really want to have a stricter contract, we should simply insist that we need the `totalPrice` and `messages` fields; if they are not present, we will return a `500` error. This is not a catastrophic failure, since our server will not crash, but it is an unforgiving approach that could induce the consumer to stop performing requests for that order for a while. If you go for this approach, you should change the setup of the success scenario to return a syntactically correct order and add new error scenarios for this.

Which approach is better should be decided on a case-by-case basis. Whenever it makes sense to assume meaningful defaults, the first one is usually better. Anyway, our web API layer defines an external contract that will be used by third parties, so you need to define a very robust implementation and be very clear about how it works in any event. That is why it is important to test what happens if the contract with the business is violated and test for all the possible scenarios in which this can happen.

Interestingly, all of these aspects are not a big problem if you do integrated tests! Since an integration test will check both layers together, any problem derived from misunderstandings between teams will be exposed. This is a clear drawback and source of complexity in the isolated testing of different abstraction layers and subsystems. However, I still think that the drawbacks of integrated testing are often bigger than the approach proposed here. After all, there will be always one point where you must break your system into pieces that need to be developed and tested independently; if not, you will end up with a big, unmaintainable monolith of code.

Finishing the scenario

To finish the scenario, we need to think about how to publish order actions and order items. The two scenarios can be as follows:

- For each one of the actions that the order can have, we can publish an extra resource with its own URI. If we access one of these "action" resources, we will get a form that can be filled and executed to perform the action.
- For each one of the items, we will also publish a resource with its URI. Performing a GET request against these resources will return the information for each one of the items.

This is a common design pattern in the world of web APIs. The idea is that each one of these resources, orders, items, and actions has links that relate them to the others. So, what we need to test at the order level is that we have the right links to these other resources.

Let's start with the items. For them, we will use the standard link type that is unsurprisingly called `item`. We need to test two cases: one is an empty order and the other is the nonempty one:

```
it('given that the order is empty, there will be no item links',
  function () {
    this.orderModel.items = [];

    return expect(this.GET(this.orderURI)).to.eventually
      .have.not.deep.property('body._links.item');
  });

it('given that the order is not empty, there will be an item link for
each item', function () {
  this.orderModel.items = ['itemX', 'itemY'];
```

```

return expect(this.GET(this.orderURI)).to.eventually
  .have.deep.property('body._links.item')
  .that.has.length(this.orderModel.items.length)
  .and.that.include.deep.members([
    {href: this.orderURI + '/item_0'},
    {href: this.orderURI + '/item_1'}
  ]);
}
);

```

Here, we are using the following design pattern for HAL documents:

- If there are no relevant links of a type, then there is no entry in the `_links` object. That is why we are asserting that there will be no `item` field in the `_links`.
- If the document can have one or more links of the same type, then the corresponding entry in the `_links` object will be an array of links. This is the case when we have one or more items.
- If there can be only one link of a type, then we do not use an array and directly put the link object. This is the case of the `self` link, as we saw in the `will have a self link` test.

The previous test is explicit about the URI for each one of the items. It must follow the `/orders/:orderId/item:_itemIndex` pattern. Also, note that the items we are using in the second test are totally fake and have nothing to do with a real order item. However, to test this functionality, the correctness of the items is not really important!



The relation between a collection and its items is being standardized at <http://tools.ietf.org/html/rfc6573>.



What about actions? We should only include links to form resources that represent actions that act directly on this order, such as `append-beverage` and `place-order`. However, actions about deletion on changing order items should belong to the item resource itself.

Where to put a link to another resource is a technical decision to be taken while testing the web API. Here, I am following my own experience: each action should be linked with the main resource it acts on. Testing the web API can generate some feedback about the design of the interface and the contract of the business layer. For example, did the business layer team do a good job putting all the actions at the order level? What do you think?

How can we model the append-beverage action? According to <http://tools.ietf.org/html/rfc6861>, the `create-form` link type is the correct choice, since it represents a link to a form that, when executed, will add a new item to the parent resource; in this case, the item is the order. Let's write some tests for this:

```
it('given that the order has not an append-beverage action,' +
  ' there will not be a link to a create-form', function () {
  this.orderModel.actions = [
    { action: 'not-an-append-beverage-action' }
  ];

  return expect(this.GET(this.orderURI)).to.eventually
    .have.not.deep.property('body._links.create-form');
});

it('given that the order has an append-beverage action,' +
  ' there will be a link to a create-form', function () {
  this.orderModel.actions = [
    { action: 'append-beverage' }
  ];

  return expect(this.GET(this.orderURI)).to.eventually
    .have.deep.property('body._links.create-form')
    .that.deep.equals({
      href: this.orderURI + '/create-form'
    });
});
```

The tests are very simple. If there is an action of type `append-beverage`, we will create a link to the `create-form` link type. We are not interested in its `target` or `parameters` fields, since they are not relevant for our tests. This can make us wonder why the business layer team added a `target` field to the action, if we do not need it. Perhaps it will be better to rename `action` as `type`. As I said earlier, this is the kind of insight that arises while doing testing; this should lead to a meaningful conversation with other teams.

We can take a similar approach for the place-order form:

```
it('given that the order has not a place-order action,' +
  ' there will not be a link to a place-order-form', function () {
  this.orderModel.actions = [
    { action: 'not-a-place-order-action' }
  ];
});
```

```

    return expect(this.GET(this.orderURI)).to.eventually
      .have.not.deep.property('body._links.place-order-form');
};

it('given that the order has a place-order action,' +
  ' there will be a link to a place-order-form', function () {
this.orderModel.actions = [
  { action: 'place-order' }
];

return expect(this.GET(this.orderURI)).to.eventually
  .have.deep.property('body._links.place-order-form')
  .that.deep.equals({
    href: this.orderURI + '/place-order-form'
  });
});

```

It is the same approach; the only difference is that the link type will be `place-order-form`. There is a clear duplication here, so let's get rid of it:

```

function willHaveALinkForTheAction(example) {
  var actionPerformed = example.actionName,
    linkType = example.linkType;

  it('given that the order has not an ' + actionPerformed
+ 'action, there will not be a link to a ' + linkType, function () {
    this.orderModel.actions = [
      { action: 'not-the-' + actionPerformed + '-action' }
    ];

    return expect(this.GET(this.orderURI)).to.eventually
      .have.not.deep.property('body._links.' + linkType);
});

  it('given that the order has an ' + actionPerformed + ' action,' +
    ' there will be a link to a ' + linkType, function () {
    this.orderModel.actions = [
      { action: actionPerformed }
    ];

    return expect(this.GET(this.orderURI)).to.eventually
      .have.deep.property('body._links.' + linkType)
      .that.deep.equals({

```

```
        href: this.orderURI + '/' + linkType
    });
}
]

[actionName: 'append-beverage', linkType: 'create-form'],
[actionName: 'place-order', linkType: 'place-order-form']
].forEach(willHaveALinkForTheAction);
```

Testing slave resources

Although we have tested the main HAL document for an order, we are still not done. We have discovered that we need to create more HAL resources, order items, and actions. We need to write tests for them too.

The order actions

As we saw earlier, each action has been extracted to a form resource. How do form resources look?

Let's start with the `/orders/:orderId/place-order-form` resource. We will need a new feature for this, since accessing a form resource is a different operation from accessing the order from the consumer's point of view.

Let's create a `test/get_placeOrderForm.js` test suite:

```
'use strict';

var chai = require('chai'),
expect = chai.expect,
Q = require('q');

chai.use(require("sinon-chai"));
chai.use(require("chai-as-promised"));

describe('GET /order/:orderId/place-order-form', function () {
beforeEach(function () {
this.orderId = "<some order id>";
this.orderURI = this.ordersBaseURI + '/' +
encodeURIComponent(this.orderId);
});
});
```

```
beforeEach(function () {
  this.placeOrderFormURI = this.orderURI + '/place-order-form';
});

context('Given that the order exists', function () {
  beforeEach(function () {
    this.orderModel = {};
  });

  this.orderSystem.display
    .withArgs(this.orderId)
    .returns(Q.fulfill(this.orderModel));
});

context('and that there is a place-order action', function() {
  beforeEach(function () {
    this.orderModel.actions = [
      { action: 'place-order' }
    ];
    this.response = this.GET(this.placeOrderFormURI);
  });
  it('will respond with a 200 code', function () {
    return expect(this.response).to.eventually
      .have.property('status', 200);
  });
  describe('will respond with a HAL document for the form',
    function () {
      it('will have a self link', function () {
        return expect(this.response).to.eventually
          .have.deep.property('body._links.self')
          .that.is.deep.equal({
            href: this.placeOrderFormURI
          });
      });
      it('will have the parent order as a target', function () {
        return expect(this.response).to.eventually
          .have.deep.property('body._links.target')
          .that.is.deep.equal({
            href: this.orderURI
          });
      });
      it('will use the POST method when submitted', function() {
        return expect(this.response).to.eventually
          .have.deep.property('body.method', 'POST');
      });
    });
});
```

```
it('will have a name property with value "place-order-form"',  
  function() {  
    return expect(this.response).to.eventually  
      .have.deep.property('body.name',  
        'place-order-form');  
  });  
  
it('will have a status parameter with value "placed"',  
  function() {  
    return expect(this.response).to.eventually  
      .have.deep.property('body.parameters')  
      .that.is.deep.equal({status: 'placed'});  
  });  
});  
});  
});  
});  
});  
});  
});  
});  
});
```

This is the success scenario: the order exists, and it contains a place order action. In this case, it needs to return a 200 code with a proper HAL representation of the form. This is more or less the same approach we used to test the order.

The main difference is in the setup. We need to first assert that the order exists. This is necessary because the `place-order-form` link type is a slave resource; in other words, it cannot exist if it is not associated with an existing order. This is emphasized by the fact that the URI of the form is an extra segment under the order's URI. To set this up, we just copied the `beforeEach()` blocks of the `get_order.js` test suite. We added an extra `beforeEach()` method to set up the forms' URI in the `this.placeOrderFormURI` field. After that, we needed to assert that the order contains a `place-order` action. Note that we did this in a separate `context()` block; I will explain in a moment why I did it in this way.

The tests themselves are not really complex. We just need to check whether the `self` and `target` links are OK, whether the HTTP method to be used with the form submission is `POST`, and whether we have the expected `name` and `status` parameters for the form.



A common design pattern to model forms in HAL is to use a `target` link and a `method` parameter to specify to which URI, and with which HTTP method, we must submit the form. Extra form parameters go inside the `parameters` property.

Now, we need to focus on the unsuccessful scenarios. For example, what happens when the order exists but it has no actions property? Or what if we have an actions property, but we don't really have a place-order action in it? If we decide that we should return a 404 error in these cases, then we can write the following test:

```
context('Given that the order exists', function () {
  beforeEach(function () {
    this.orderModel = {};

    this.orderSystem.display
      .withArgs(this.orderId)
      .returns(Q.fulfill(this.orderModel));
  });

  it('and that there is no actions property, will respond with a
     404 code', function () {
    return expect(this.GET(this.placeOrderFormURI)).to.eventually
      .have.property('status', 404);
  });
  context('and that there is a place-order action', function () {
    // Skipped for brevity
  });

  it('and that there is no place-order action, will respond with a
     404 code', function () {
    this.orderModel.actions = [
      { action: 'not-a-place-order-action' }
    ];

    return expect(this.GET(this.placeOrderFormURI)).to.eventually
      .have.property('status', 404);
  });
});
});
```

Now we can see why I defined an extra nested `context()` block for the success case. This way, we can reuse the setup of the order between the following three scenarios:

- The order exists, and there is no actions property
- The order exists, and there is an actions property with a place-order action
- The order exists, and there is an actions property without a place-order action

Finally, we can create two more error scenarios: the order does not exist and the business layer generates an error:

```
context('Given that the order does not exists', function () {
  beforeEach(function () {
    this.orderSystem.display
      .withArgs(this.orderId)
      .returns(Q.fulfill(null));
  });
  it('will respond with a 404 code', function () {
    return expect(this.GET(this.placeOrderFormURI)).to.eventually
      .have.property('status', 404);
  });
});

context('Given that the order subsystem is down', function () {
  beforeEach(function () {
    this.orderSystem.display
      .withArgs(this.orderId)
      .returns(Q.reject(new Error('Expected error')));
  });
  it('will respond with a 500 code', function () {
    return expect(this.GET(this.placeOrderFormURI)).to.eventually
      .have.property('status', 500);
  });
});
```

These scenarios are exactly the same as the ones in the order resource, except that they try to get the place-order-form URI.

We can model the create-form resource in a similar way. We can create test/get_createForm.js with the following lines of code:

```
'use strict';

var chai = require('chai'),
  expect = chai.expect,
  Q = require('q');

chai.use(require("sinon-chai"));
chai.use(require("chai-as-promised"));

describe('GET /order/:orderId/create-form', function () {
  beforeEach(function () {
```

```
// Skipped for brevity
});

beforeEach(function () {
    this.createFormURI = this.orderURI + '/create-form';
});

context('Given that the order exists', function () {
    beforeEach(function () {
        // Skipped for brevity
    });

    it('and that there is no actions property, will respond with a
       404 code', function () {
        return expect(this.GET(this.createFormURI)).to.eventually
            .have.property('status', 404);
    });
});

context('and that there is a append-beverage action', function () {
    beforeEach(function () {
        this.orderModel.actions = [
            { action: 'append-beverage' }
        ];
    });

    it('will respond with a 200 code', function () {
        return expect(this.GET(this.createFormURI)).to.eventually
            .have.property('status', 200);
    });
});

describe('will respond with a HAL document for the form',
    function () {
        it('will have a self link', function () {
            return expect(this.GET(this.createFormURI)).to.eventually
                .have.deep.property('body._links.self')
                .that.is.deep.equal({
                    href: this.createFormURI
                });
        });
    });
});

it('will use the PUT method when submitted', function () {
```

```
        return expect(this.GET(this.createFormURI)).to.eventually
            .have.deep.property('body.method', 'PUT');
    });

    it('will have a name property with value "create-form"',
        function () {
            return expect(this.GET(this.createFormURI)).to.eventually
                .have.deep.property('body.name', 'create-form');
        });
});

function appendBeverageActionWithParametersScenario(example) {
    context('and the action has ' + example.description, function
() {
    beforeEach(function () {
        this.orderModel.actions[0].parameters = {
            beverageRef: example.beverageRef,
            quantity: example.quantity
        };

        this.response = this.GET(this.createFormURI);
    });

    it('the form will have a beverageHref parameter with the
        URI of the specified beverage', function () {
        return expect(this.response).to.eventually.have.deep
            .property('body.parameters.beverageHref',
                example.expectedBeverageURI);
    });

    it('the form will have a quantity parameter with the
        specified quantity', function () {
        return expect(this.response).to.eventually.have.deep
            .property('body.parameters.quantity',
                example.expectedQuantity);
    });
});
}

[
{
    description: 'no default beverage',
    quantity: 10, beverageRef: null,
```

```

        expectedQuantity: 10, expectedBeverageURI: null
    },
    {
        description: 'a default beverage',
        quantity: 2, beverageRef: '<some beverage>',
        expectedQuantity: 2, expectedBeverageURI:
            '/beverages/%3Csome%20beverage%3E'
    }
].forEach(appendBeverageActionWithParametersScenario);

function appendBeverageFormTargetsNewItemURI(example) {
    it('and the order has ' + example.items.length +
        " items, the target of the form will point to " +
        example.expectedTarget, function () {
        this.orderModel.items = example.items;

        return expect(this.GET(this.createFormURI)).to.eventually
            .have.deep.property('body._links.target')
            .that.is.deep.equal({
                href: this.orderURI + example.expectedTarget
            });
    });
}

[
    {items: ['item0', 'item1'], expectedTarget: '/item_2'},
    {items: [], expectedTarget: '/item_0'}
].forEach(appendBeverageFormTargetsNewItemURI);
});

it('and that there is no append-beverage action, will respond
    with a 404 code', function () {
    // Skipped for brevity
});

context('Given that the order does not exists', function () {
    // Skipped for brevity
});

context('Given that the order subsystem is down', function () {
    // Skipped for brevity
});
}

```

I omitted the failure scenarios and the common setup, since they are very similar to the ones in `place-order-form`. The interesting case here is the success-case scenario that has been divided in to three different scenarios, which are as follows:

- The first scenario is like the one we have for `place-order-form`; it checks the `method`, `name`, and `self` links, and so on.
- The second one is a parameterized scenario that will be executed with different examples of the parameter values that this action can have. This will allow us to cover cases such as what would happen if the `beverageRef` were null.
- The third one is about the `target` URI. It is a common technique to design `create-form` to use `PUT` instead of `POST`. This is better because the `PUT` method is idempotent and, if the consumer issues a duplicated request, there will not be an accidental item creation. The problem is that the `target` URI must point to the URI of the new item, and this will change with the number of items the order has.

Testing embedded resources

Until now, we have been representing the relationship between resources using links. Although this is correct, sometimes, it can be not very performant, since the consumer will need several network calls to retrieve all the resources it is interested in. For this reason, it is normal to provide the most useful slave resources embedded in the main document.

In the case of our order, it would be interesting to embed its item resources and its forms. In HAL, we can use the `_embedded` section for this purpose. However, how do we test it? We should test that the `_embedded` section contains the correct item and form resource, but this implies that we need to repeat the tests we already have for those resources themselves. It is clear that we need to extract these tests to some reusable module. Let's create a `test/specs/createForm.js` file with the following contents:

```
'use strict';

var chai = require('chai'),
    expect = chai.expect;

module.exports = {
  willBeACreateForm: function () {
    it('will have a self link', function () {
```

```
return expect(this.createForm).to.eventually
    .have.deep.property('_links.self')
    .that.is.deep.equal({
        href: this.createFormURI
    });
});

it('will use the PUT method when submitted', function () {
    return expect(this.createForm).to.eventually
        .have.property('method', 'PUT');
});

it('will have a name property with value "create-form"', function () {
    return expect(this.createForm).to.eventually
        .have.property('name', 'create-form');
},
willHaveTheRightParameters: function (example) {
    it('the form will have a beverageHref parameter with the URI of
the specified beverage', function () {
        return expect(this.createForm).to.eventually.have.deep
            .property('parameters.beverageHref',
                example.expectedBeverageURI);
    });

    it('the form will have a quantity parameter with the
specified quantity', function () {
        return expect(this.createForm).to.eventually.have.deep
            .property('parameters.quantity', example.expectedQuantity);
    },
    willHaveTheRightTarget: function (example) {
        it('the target of the form will point to ' +
            example.expectedTarget, function () {
            return expect(this.createForm).to.eventually
                .have.deep.property('_links.target')
                .that.is.deep.equal({
                    href: this.orderURI + example.expectedTarget
                });
        });
    }
});
};
```

This new module will contain all the tests about what is a correct `create-form` resource for an order. They are separated in different utility methods, each one containing the tests for the different success scenarios we had in the `get_createForm.js` test suite. There is a minor difference in the code; we are now testing against `this.createForm`. This field must be set up previously with a promise of the `create-form` object.

We can change `test/get_createForm.js` to use this module:

```
'use strict';

var chai = require('chai'),
    expect = chai.expect,
    Q = require('q'),
    acreateForm = require('../specs/createForm');

chai.use(require("sinon-chai"));
chai.use(require("chai-as-promised"));

describe('GET /order/:orderId/create-form', function () {
    // Skipped for brevity
    context('Given that the order exists', function () {
        // Skipped for brevity
        context('and that there is an append-beverage action', function () {
            // Skipped for brevity
            describe('will respond with a HAL document for the form',
                function () {
                    beforeEach(function () {
                        // Store create-form in this.createForm
                        // This way it can be accessed by the embedded tests
                        this.createForm = this
                            .GET(this.createFormURI)
                            .then(function (response) {
                                // The create-form is in the body of the response
                                return response.body;
                            });
                    });
                    // This will embed the tests for a create-form here
                    acreateForm.willBeACreateForm();
                });
            // Skipped for brevity
        });
        // Skipped for brevity
    });
    // Skipped for brevity
});
```

We just need to import the module and call it in the appropriate places. Each scenario is defined with a corresponding `context()` block. Inside these blocks, there is a `beforeEach()` method that performs the required setup, makes the GET request, and then extracts the `create-form` action from the body of the response. This will result in a promise that contains the form itself. This promise is stored in `this.createForm`, so it can be accessible from the embedded tests. Then, we just simply invoke the methods of `test/specs/createForm.js` to embed the relevant tests inside this feature.

Now we can use the same approach in `test/get_createForm.js` to add scenarios that test that the form has been created with the right parameters and has the correct target property:

```
context('and that there is an append-beverage action', function () {
    // Skipped for brevity
    function appendBeverageActionWithParametersScenario(example) {
        context('and the action has ' + example.description, function () {
            beforeEach(function () {
                this.orderModel.actions[0].parameters = {
                    beverageRef: example.beverageRef,
                    quantity: example.quantity
                };

                this.createForm = this
                    .GET(this.createFormURI)
                    .then(function (response) {
                        return response.body;
                    });
            });

            aCreateForm.willHaveTheRightParameters(example);
        });
    }
}

[
{
    description: 'no default beverage',
    quantity: 10, beverageRef: null,
    expectedQuantity: 10, expectedBeverageURI: null
},
{
    description: 'a default beverage',
    quantity: 2, beverageRef: '<some beverage>',
}
```

```
    expectedQuantity: 2, expectedBeverageURI:
      '/beverages/%3Csome%20beverage%3E'
    }
  ].forEach(appendBeverageActionWithParametersScenario);

  function appendBeverageFormTargetsNewItemURI(example) {
    context('given the order has ' + example.items.length + ' items',
    function () {
      beforeEach(function () {
        this.orderModel.items = example.items;
        this.createForm = this
          .GET(this.createFormURI)
          .then(function (response) {
            return response.body;
          });
      });
      aCreateForm.willHaveTheRightTarget(example);
    });
  }

  [
    {items: ['item0', 'item1'], expectedTarget: '/item_2'},
    {items: [], expectedTarget: '/item_0'}
  ].forEach(appendBeverageFormTargetsNewItemURI);
});
```

Now we can use the same approach in `test/get_order.js` to add a new scenario that describes how a `create-form` resource is embedded in the order:

```
describe('will embed the create-form resource', function () {
  beforeEach(function () {
    this.orderModel.actions = [
      { action: 'append-beverage' }
    ];
  });

  it('will have an embedded resource named create-form', function () {
    return expect(this.GET(this.orderURI)).to.eventually
      .have.deep.property('body._embedded.create-form')
  });

  describe('the embedded resource will be a valid create-form',
  function () {
    var scenarioParameters = {
```

```

        items: ['itemX', 'itemY', 'itemZ'], expectedTarget: '/item_3',
        quantity: 3, beverageRef: '<some other beverage>',
        expectedQuantity: 3, expectedBeverageURI: '/beverages/%3Csome%20
other%20beverage%3E'
    };
    beforeEach(function () {
        this.createFormURI = this.orderURI + '/create-form';
        this.orderModel.items = scenarioParameters.items;
        this.orderModel.actions[0].parameters = {
            beverageRef: scenarioParameters.beverageRef,
            quantity: scenarioParameters.quantity
        };
        this.createForm = this
            .GET(this.orderURI)
            .then(function (response) {
                if (response.body && response.body._embedded)
                    return response.body._embedded['create-form'];
            });
    });
    aCreateForm.willBeACreateForm();
    aCreateForm.willHaveTheRightParameters(scenarioParameters);
    aCreateForm.willHaveTheRightTarget(scenarioParameters);
});
});

```

In this case, it is a bit simpler. I expect that the logic that generates the `create-form` resource is going to be reused to embed it into the order. It does not make much sense either to test it so exhaustively or to triangulate it, because this logic has already been tested in `test/get_createForm.js`.

In a similar way, we can create `specs/placeOrderForm.js` as follows:

```

'use strict';

var chai = require('chai'),
    expect = chai.expect;

module.exports = function () {
    it('will have a self link', function () {
        return expect(this.placeOrderForm).to.eventually
            .have.deep.property('_links.self')
            .that.is.deep.equal({
                href: this.placeOrderFormURI
            });
    });
}

```

```
}) ;

it('will have the parent order as a target', function () {
    return expect(this.placeOrderForm).to.eventually
        .have.deep.property('_links.target')
        .that.is.deep.equal({
            href: this.orderURI
        });
});
it('will use the POST method when submitted', function () {
    return expect(this.placeOrderForm).to.eventually
        .have.property('method', 'POST');
});

it('will have a name property with value "place-order-form"',
    function () {
    return expect(this.placeOrderForm).to.eventually
        .have.property('name', 'place-order-form');
});

it('will have a status parameter with value "placed"', function () {
    return expect(this.placeOrderForm).to.eventually
        .have.property('parameters')
        .that.is.deep.equal({status: 'placed'});
});
};
```

The only difference is that, because we have only a single scenario, we export a single function.

Extracting cross-cutting scenarios

As you may have noticed, in all the `get_*.js` test suites there will always be a scenario to test what happens when an order does not exist, what happens when the `display()` method throws an error, and so on. In these cases, the only difference is in the exact URI we are using in the GET request, but the result and setup are the same across all the test suites.

We can create a new feature to capture all of these *bad-weather* scenarios in `test/get_resource_fails.js`:

```
'use strict';

var chai = require('chai'),
```

```
expect = chai.expect,
Q = require('q');

chai.use(require("sinon-chai"));
chai.use(require("chai-as-promised"));

function commonFailureScenarios(example) {
  describe('GET ' + example.resource + ' fails:', function () {
    context('The order does not exists', function () {
      beforeEach(function () {
        this.orderSystem.display
          .withArgs(example.orderId)
          .returns(Q.fulfill(null));

        this.response = this.GET(this.ordersBaseURI + example.uri);
      });
      it('will respond with a 404 code', function () {
        return expect(this.response).to.eventually
          .have.property('status', 404);
      });
    });
  });

  context('The order subsystem is down', function () {
    beforeEach(function () {
      this.orderSystem.display
        .withArgs(example.orderId)
        .returns(Q.reject(new Error('Expected error')));
    });

    this.response = this.GET(this.ordersBaseURI + example.uri);
  });
  it('will respond with a 500 code', function () {
    return expect(this.response).to.eventually
      .have.property('status', 500);
  });
});
});

[
{
  resource: "/order/:orderId",
  orderId: "<some order id>",


```

```
        uri: '/%3Csome%20order%20id%3E'
    },
{
    resource: "/order/:orderId/create-form",
    orderId: "<some order>",
    uri: '/%3Csome%20order%3E/create-form'
},
{
    resource: "/order/:orderId/place-order-form",
    orderId: "order-id",
    uri: '/order-id/place-order-form'
}
].forEach(commonFailureScenarios);
```

This will create a test suite for each order-related resource, with a scenario for each common cause of failure. Now we can just simply remove all those *bad-weather* scenarios from all the test suites. For example, we can simplify `get_placeOrderForm.js` in the following way:

```
describe('GET /order/:orderId/place-order-form', function () {
    beforeEach(function () {
        this.orderId = "<some order id>";
        this.orderURI = this.ordersBaseURI + '/' +
            encodeURIComponent(this.orderId);
        this.placeOrderFormURI = this.orderURI + '/place-order-form';

        this.orderModel = {};
        this.orderSystem.display
            .withArgs(this.orderId)
            .returns(Q.fulfill(this.orderModel));
    });

    it('and that there is no actions property, will respond with a
       404 code', function () {
        return expect(this.GET(this.placeOrderFormURI)).to.eventually
            .have.property('status', 404);
    });

    context('and that there is a place-order action', function () {
        beforeEach(function () {
            this.orderModel.actions = [
                { action: 'place-order' }
            ];
            this.response = this.GET(this.placeOrderFormURI);
        });

        it('will respond with a 200 code', function () {
            return expect(this.response).to.eventually
                .have.property('status', 200);
        });
    });
});
```

```
});
describe('will respond with a HAL document for the form',
  function () {
    beforeEach(function () {
      this.placeOrderForm = this.response.then(function (response) {
        return response.body;
      });
    });
    isAPlaceOrderForm();
  });
});

it('and that there is no place-order action, will respond with a
404 code', function () {
  this.orderModel.actions = [
    { action: 'not-a-place-order-action' }
  ];

  return expect(this.GET(this.placeOrderFormURI)).to.eventually
    .have.property('status', 404);
});
});
```

Note that the test suite has been dramatically simplified! We can do the same for the `get_createForm.js` and `get_order.js` resources.

Homework!

Our API is far from done; there are still a lot of features to test and implement in our API. For example, we could add tests to embed `place-order-form` into the order. However, there is a lot of more to do.

In the case of form resources, we will need to add new features to show how to fill and execute them. Another thing that is missing is a test for item resources, both in the standalone and embedded modes. For brevity, and as we do not have the corresponding operations on the business layer for some of those features, I will leave it as an exercise for you. Some hints are as follows:

- Items are slave resources, so expect scenarios similar to the actions.
- Items can have actions too, such as editing the quantity and deleting them from the order. You will need to create form resources for them and test them.
- An item should relate somehow to the beverage it refers to. Obviously, there should be a beverage subsystem with its own API, so you can simply define a reference to the URI of the relevant beverage.

Until now, we have been focusing only on how to read the state of the order resource and its slaves – in other words, the read-only functionality of the web API. We also need to test the write capabilities of the API. In a well-designed web API, we should only try to change the state of a resource executing the forms that we return as part of the representation of that resource. Executing a form means sending a request to the URI specified on `target` using the value specified in the `method` field. As the body in the request, we should send the data contained in the form's parameters. You can test a form execution with the techniques you already learned in this chapter. Just follow the following structure for your features:

- The setup is made as in the rest of the tests seen earlier, using the test double of the business layer.
- To drive the test execution of the forms:
 - Issue a request to the `target` URI of the form using the value of the `method` field as the HTTP method.
 - Pass in the body of the request and the parameters of the form using the appropriate mime type. The mime type, which is part of the contract of your API, should be explicit in the tests. Triangulate using different parameter values.
- Assert that the correct call happens in the business layer, and check whether the parameters of the call are consistent with the request's body. Each form execution should have a one-to-one mapping with a specific method in the business layer. Remember that the responsibility of the web API layer is to publish the business layer, not to add extra logic to it.

What about the beverage resources? Should we create tests for them as well? We have been focusing on the business processes related to orders and how to publish them as a web API. So, I think that the beverages probably belong to another subsystem, maybe inventory, so it is better to leave the tests for that subsystem. However, this kind of decision always depends a lot on the specifics of each business.

As you can see, the problem domain of publishing business functionality as a web API is not small. Imagine what would have happened if we had tested both layers together?

Summary

The naïve approach of testing both the web API layer and the business layer is not usually a good idea. We can run into problems such as slow tests, difficulties in debugging the tests, excessive complexity, and so on. Instead, it is better to slice the system into two layers: the web API, which is responsible for publishing the business logic over the Web, and the business layer, which is responsible for implementing the business rules themselves.

If both layers are developed by the same team, then we can assume that the business layer will always fulfill its contract. If not, it will be better to add new scenarios to check whether the web layer reacts in a sensible way if the business does not comply with its contract (that is, it has a bug).

Since the only responsibility of the web API layer is to publish the business layer over HTTP, we should not try to add any more logic here. This means that we should be agnostic about the business rules in our tests of the web API layer. So, try not to make the setup of your test more complex than needed, because you want to use the correct model.

Drive the server through a real HTTP request. Create a small utility for this; you can attach this to the runtime context of Mocha to access it easily. During setup, start and stop the server only once, but set up the server with a brand new test double of the business layer before each test.

For each of the resources, add a new feature for each HTTP verb you can use to access a resource. In particular, form resources will need additional features that show how to fill and execute them. Add a new feature for each mime type to which you plan to publish the resource.

In each of these features, check for the HTTP status code and check whether the returned body is well formed. This means that you need to at least check the body for its data fields, self link, links to related resources, and whether there are embedded resources or not.

To simplify your test codebase, extract the tests for each resource in a function that can be reused from other tests. This will remove duplication if you need to test embedded resources.

We will leave the realm of the abstract now; in the next chapter, we will explore how to test an UI. This is a complex subject indeed!

6

Testing a UI Using WebDriverJS

In this chapter, we will look into an advanced concept: how to test a user interface. For this purpose, you will learn the following topics:

- Using WebDriverJS to manipulate a browser and inspect the resulting HTML generated by our UI
- Organizing our UI codebase to make it easily testable
- The right abstraction level for our UI tests
- Testing modern, rich Internet applications

Our strategy for UI testing

There are two traditional strategies towards approaching the problem of UI testing: record-and-replay tools and end-to-end testing.

The first approach, record-and-replay, leverages the use of tools capable of recording user activity in the UI and saves this into a script file. This script file can be later executed to perform exactly the same UI manipulation as the user performed and to check whether the results are exactly the same. This approach is not very compatible with BDD because of the following reasons:

- We cannot test-first our UI. To be able to use the UI and record the user activity, we first need to have most of the code of our application in place. This is not a problem in the waterfall approach, where QA and testing are performed after the codification phase is finished. However, in BDD, we aim to document the product features as automated tests, so we should write the tests before or during the coding.

- The resulting test scripts are low-level and totally disconnected from the problem domain. There is no way to use them as a live documentation for the requirements of the system.
- The resulting test suite is brittle and it will stop working whenever we make slight changes, even cosmetic ones, to the UI. The problem is that the tools record the low-level interaction with the system that depends on technical details of the HTML.

The other classic approach is end-to-end testing, where we do not only test the UI layer, but also most of the system or even the whole of it. To perform the setup of the tests, the most common approach is to substitute the third-party systems with test doubles. Normally, the database is under the control of the development team, so some practitioners use a regular database for the setup. However, we could use an in-memory database or even mock the DAOs. In any case, this approach prompts us to create an integrated test suite where we are not only testing the correctness of the UI, but the business logic as well.



In the context of this discussion, an **integrated test** is a test that checks several layers of abstraction, or subsystems, in combination. Do not confuse it with the act of testing several classes or functions together.

This approach is not inherently against BDD; for example, we could use Cucumber.js to capture the features of the system and implement Gherkin steps using WebDriver to drive the UI and make assertions. In fact, for most people, when you say BDD they always interpret this term to refer to this kind of test.

Unfortunately, as we saw in the previous chapter, integrated tests are not a good idea. We will end up writing a lot of test cases, because we need to combine the scenarios from the business logic domain with the ones from the UI domain. Furthermore, in which language should we formulate the tests? If we use the UI language, maybe it will be too low-level to easily describe business concepts. If we use the business domain language, maybe we will not be able to test the important details of the UI because they are too low-level. Alternatively, we can even end up with tests that mix UI language with business terminology, so they will neither be focused nor very clear to anyone.

Choosing the right tests for the UI

If we want to test whether the UI works, why should we test the business rules?

After all, this is already tested in the BDD test suite of the business logic layer. To decide which tests to write, we should first determine the responsibilities of the UI layer, which are as follows:

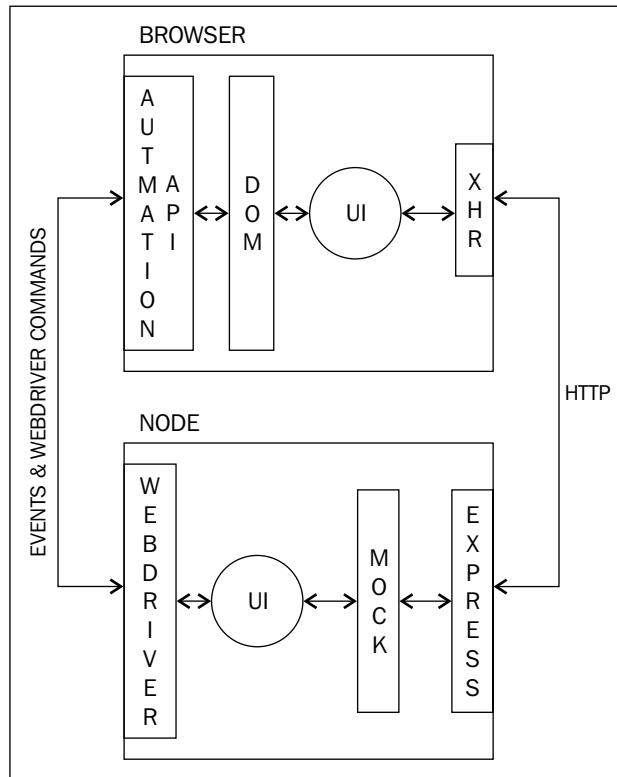
- Presenting the information provided by the business layer to the user in a nice way.
- Transforming user interaction into requests for the business layer.
- Controlling the changes in the appearance of the UI components, which includes things such as enabling/disabling controls, highlighting entry fields, showing/hiding UI elements, and so on.
- Orchestration between the UI components. Transferring and adapting information between the UI components and navigation between pages fall under this category.

Thus, we are in a similar situation compared to the one in the previous chapter. We do not need to write tests about business rules, and we should not assume much about the business layer itself, apart from a loose contract. In this sense, all the advice of the previous chapter is still valid for testing a UI.

How we should word our tests? We should use a UI-related language when we talk about what the user sees and does. Words such as fields, buttons, forms, links, click, hover, highlight, enable/disable, or show and hide are relevant in this context. However, we should not go too far; otherwise, our tests will be too brittle. Saying, for example, that the name field should have a pink border is too low-level. The moment that the designer decides to use red instead of pink, or changes his mind and decides to change the background color instead of the border, our test will break. We should aim for tests that express the real intention of the user interface; for example, the name field should be highlighted as incorrect.

The testing architecture

At this point, we could write tests relevant for our UI using the following testing architecture:



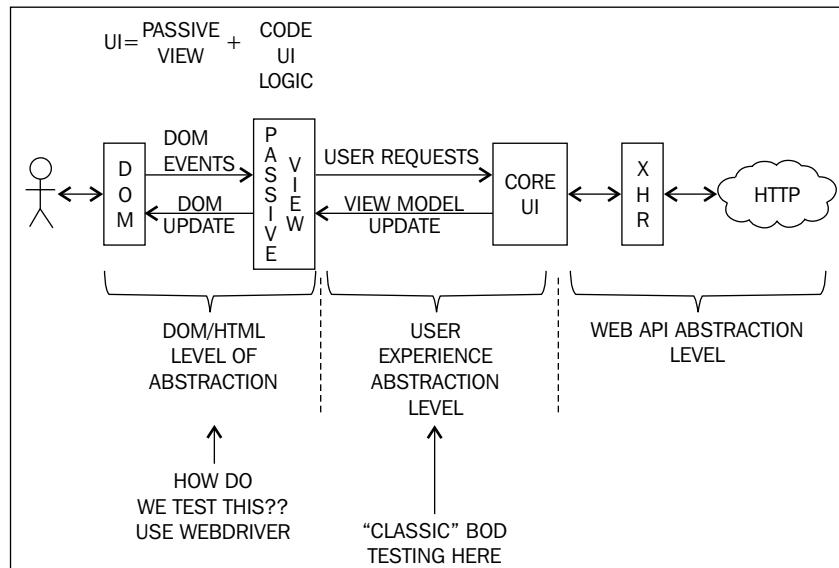
A simple testing architecture for our UI

We can use WebDriver to issue user gestures to interact with the browser. These user gestures are transformed by the browser into DOM events that are the inputs of our UI logic and will trigger operations on it. We can use WebDriver again to read the resulting HTML in the assertions. We can simply use a test double to impersonate our server, so we can set up our tests easily.

This architecture is very simple and sounds like a good plan, but it is not! There are three main problems here:

- UI testing is very slow. Take into account that the boot time and shutdown phase can take 3 seconds in a normal laptop. Each UI interaction using WebDriver can take between 50 and 100 milliseconds, and the latency with the fake server can be an extra 10 milliseconds. This gives us only around 10 tests per second, plus an extra 3 seconds.
- UI tests are complex and difficult to diagnose when they fail. What is failing? Our selectors used to tell WebDriver how to find the relevant elements. Some race condition we were not aware of? A cross-browser issue? Also note that our test is now distributed between two different processes, a fact that always makes debugging more difficult.
- UI tests are inherently brittle. We can try to make them less brittle with best practices, but even then a change in the structure of the HTML code will sometimes break our tests. This is a bad thing because the UI often changes more frequently than the business layer.

As UI testing is very risky and expensive, we should try to code as less amount of tests that interact with the UI as possible. We can achieve this without losing testing power, with the following testing architecture:



A smarter testing architecture

We have now split our UI layer into two components: the view and the UI logic.

This design aligns with the family of MV* design patterns. In the context of this chapter, the view corresponds with a passive view, and the UI logic corresponds with the controller or the presenter, in combination with the model. A passive view is usually very hard to test; so in this chapter we will focus mostly on how to do it. You will often be able to easily separate the passive view from the UI logic, especially if you are using an MV* pattern, such as MVC, MVP, or MVVM.

Most of our tests will be for the UI logic. This is the component that implements the client-side validation, orchestration of UI components, navigation, and so on. It is the UI logic component that has all the rules about how the user can interact with the UI, and hence it needs to maintain some kind of internal state.

The UI logic component can be tested completely in memory using standard techniques, such as the ones we saw in the early chapters of this book. We can simply mock the XMLHttpRequest object, or the corresponding object in the framework we are using, and test everything in memory using a single Node.js process. No interaction with the browser and the HTML is needed, so these tests will be blazingly fast and robust.

Then we need to test the view. This is a very thin component with only two responsibilities:

- Manipulating and updating the HTML to present the user with the information whenever it is instructed to do so by the UI logic component
- Listening for HTML events and transforming them into suitable requests for the UI logic component

The view should not have more responsibilities, and it is a stateless component. It simply does not need to store the internal state, because it only transforms and transmits information between the HTML and the UI logic. Since it is the only component that interacts with the HTML, it is the only one that needs to be tested using WebDriver.

The point of all of this is that the view can be tested with only a bunch of tests that are conceptually simple. Hence, we minimize the number and complexity of the tests that need to interact with the UI.

WebDriverJS

Testing the passive view layer is a technical challenge. We not only need to find a way for our test to inject native events into the browser to simulate user interaction, but we also need to be able to inspect the DOM elements and inject and execute scripts. This was very challenging to do approximately 5 years ago. In fact, it was considered complex and expensive, and some practitioners recommended not to test the passive view. After all, this layer is very thin and mostly contains the bindings of the UI to the HTML DOM, so the risk of error is not supposed to be high, specially if we use modern cross-browser frameworks to implement this layer.

Nonetheless, nowadays the technology has evolved, and we can do this kind of testing without much fuss if we use the right tools. One of these tools is Selenium 2.0 (also known as WebDriver) and its library for JavaScript, which is WebDriverJS (<https://code.google.com/p/selenium/wiki/WebDriverJs>).



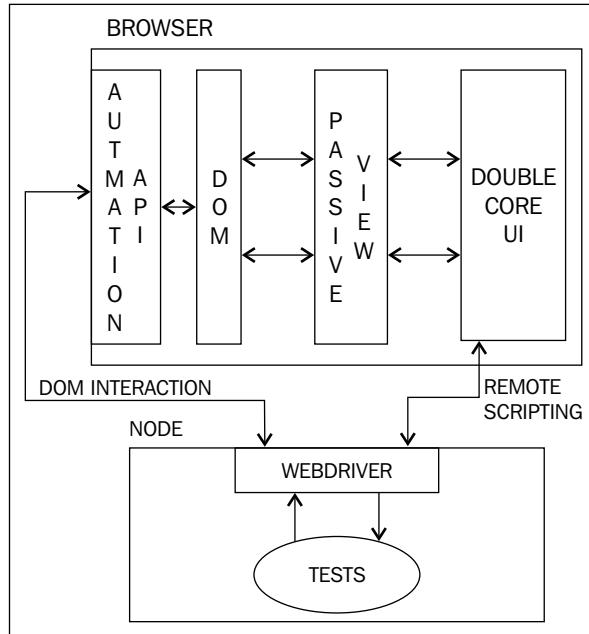
In this book, we will use WebDriverJS, but there are other bindings in JavaScript for Selenium 2.0, such as WebDriverIO (<http://webdriver.io/>). You can use the one you like most or even try both. The point is that the techniques I will show you here can be applied with any client of WebDriver or even with other tools that are not WebDriver.

Selenium 2.0 is a tool that allows us to make direct calls to a browser automation API. This way, we can simulate native events, we can access the DOM, and we can control the browser. Each browser provides a different API and has its own quirks, but Selenium 2.0 will offer us a unified API called the WebDriver API. This allows us to interact with different browsers without changing the code of our tests. As we are accessing the browser directly, we do not need a special server, unless we want to control browsers that are on a different machine.



Actually, this is only true, due some technical limitations, if we want to test against a Google Chrome or a Firefox browser using WebDriverJS. For any other browser, we are forced to use the Selenium Server, as we will see in *Chapter 8, Testing in Several Browsers with Protractor and WebDriver*.

So, basically, the testing architecture for our passive view looks like this:



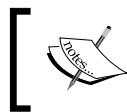
Testing with WebDriverJS

We can see that we use WebDriverJS for the following:

- Sending native events to manipulate the UI, as if we were the user, during the action phase of our tests
- Inspecting the HTML during the assert phase of our test
- Sending small scripts to set up the test doubles, check them, and invoke the `update` method of our passive view

Apart from this, we need some extra infrastructure, such as a web server that serves our test HTML page and the components we want to test.

As is evident from the diagram, the commands of WebDriverJS require some network traffic to able to send the appropriate request to the browser automation API, wait for the browser to execute, and get the result back through the network. This forces the API of WebDriverJS to be asynchronous in order to not block unnecessarily. That is why WebDriverJS has an API designed around promises. Most of the methods will return a promise or an object whose methods return promises. This plays perfectly well with Mocha and Chai.



There is a W3C specification for the WebDriver API. If you want to have a look, just visit <https://dvcs.w3.org/hg/webdriver/raw-file/default/webdriver-spec.html>.



The API of WebDriverJS is a bit complex, and you can find its official documentation at http://selenium.googlecode.com/git/docs/api/javascript/module_selenium-webdriver.html. However, to follow this chapter, you do not need to read it, since I will now show you the most important API that WebDriverJS offers us.

Finding and interacting with elements

It is very easy to find an HTML element using WebDriverJS; we just need to use either the `findElement` or the `findElements` methods. Both methods receive a locator object specifying which element or elements to find. The first method will return the first element it finds, or simply fail with an exception, if there are no elements matching the locator. The `findElements` method will return a promise for an array with all the matching elements. If there are no matching elements, the promised array will be empty and no error will be thrown.

How do we specify which elements we want to find? To do so, we need to use a locator object as a parameter. For example, if we would like to find the element whose identifier is `order_item1`, then we could use the following code:

```
var By = require('selenium.webdriver').By;

driver.findElement(By.id('order_item1'));
```

We need to import the `selenium.webdriver` module and capture its locator factory object. By convention, we store this locator factory in a variable called `By`. Later, we will see how we can get a WebDriverJS instance.

This code is very expressive, but a bit verbose. There is another version of this:

```
driver.findElement({ id: 'order_item1' });
```

Here, the locator criteria is passed in the form of a plain JSON object. There is no need to use the `By` object or any factory. Which version is better? Neither. You just use the one you like most. In this chapter, the plain JSON locator will be used.

The following are the criteria for finding elements:

- Using the tag name, for example, to locate all the `` elements in the document:

```
driver.findElements(By.tagName('li'));
driver.findElements({ tagName: 'li' });
```
- We can also locate using the `name` attribute. It can be handy to locate the input fields. The following code will locate the first element named `password`:

```
driver.findElement(By.name('password'));
driver.findElement({ name: 'password' });
```
- Using the class name; for example, the following code will locate the first element that contains a class called `item`:

```
driver.findElement(By.className('item'));
driver.findElement({ className: 'item' });
```
- We can use any CSS selector that our target browser understands. If the target browser does not understand the selector, it will throw an exception; for example, to find the second item of an order (assuming there is only one order on the page):

```
driver.findElement(By.css('.order .item:nth-of-type(2)'));
driver.findElement({ css: '.order .item:nth-of-type(2)' });
```

Using only the CSS selector you can locate any element, and it is the one I recommend. The other ones can be very handy in specific situations.

There are more ways of locating elements, such as `linkText`, `partialLinkText`, or `xpath`, but I seldom use them. Locating elements by their text, such as in `linkText` or `partialLinkText`, is brittle because small changes in the wording of the text can break the tests. Also, locating by `xpath` is not as useful in HTML as using a CSS selector. Obviously, it can be used if the UI is defined as an XML document, but this is very rare nowadays.

In both methods, `findElement` and `findElements`, the resulting HTML elements are wrapped as a `WebElement` object. This object allows us to send an event to that element or inspect its contents. Some of its methods that allow us to manipulate the DOM are as follows:

- `clear()`: This will do nothing unless `webElement` represents an input control. In this case, it will clear its value and then trigger a `change` event. It returns a promise that will be fulfilled whenever the operation is done.

- `sendKeys(text or key, ...)`: This will do nothing unless `WebElement` is an input control. In this case, it will send the equivalents of keyboard events to the parameters we have passed. It can receive one or more parameters with a text or key object. If it receives a text, it will transform the text into a sequence of keyboard events. This way, it will simulate a user typing on a keyboard. This is more realistic than simply changing the `value` property of an input control, since the proper `keyDown`, `keyPress`, and `keyUp` events will be fired. A promise is returned that will be fulfilled when all the key events are issued. For example, to simulate that a user enters some search text in an input field and then presses *Enter*, we can use the following code:

```
var Key = require('selenium-webdriver').Key;

var searchField = driver.findElement({name: 'searchTxt'});
searchField.sendKeys('BDD with JS', Key.ENTER);
```

 The `webdriver.Key` object allows us to specify any key that does not represent a character, such as *Enter*, the up arrow, *Command*, *Ctrl*, *Shift*, and so on. We can also use its `chord` method to represent a combination of several keys pressed at the same time. For example, to simulate *Alt + Command + J*, use `driver.sendKeys(Key.chord(Key.ALT, Key.COMMAND, 'J'))`.

- `click()`: This will issue a click event just in the center of the element. The returned promise will be fulfilled when the event is fired.

 Sometimes, the center of an element is nonclickable, and an exception is thrown! This can happen, for example, with table rows, since the center of a table row may just be the padding between cells!

- `submit()`: This will look for the form that contains this element and will issue a submit event.

Apart from sending events to an element, we can inspect its contents with the following methods:

- `getId()`: This will return a promise with the internal identifier of this element used by WebDriver. Note that this is not the value of the DOM `ID` property!
- `getText()`: This will return a promise that will be fulfilled with the *visible* text inside this element. It will include the text in any child element and will trim the leading and trailing whitespaces. Note that, if this element is not displayed or is hidden, the resulting text will be an empty string!

- `getInnerHtml()` and `getOuterHtml()`: These will return a promise that will be fulfilled with a string that contains `innerHTML` or `outerHTML` of this element.
- `isSelected()`: This will return a promise with a Boolean that determines whether the element has either been selected or checked. This method is designed to be used with the `<option>` elements.
- `isEnabled()`: This will return a promise with a Boolean that determines whether the element is enabled or not.
- `isDisplayed()`: This will return a promise with a Boolean that determines whether the element is displayed or not. Here, "displayed" is taken in a broad sense; in general, it means that the user can see the element without resizing the browser. For example, whether the element is hidden, whether it has `display: none`, or whether it has no size, or is in an inaccessible part of the document, the returned promise will be fulfilled as `false`.
- `getTagName()`: This will return a promise with the tag name of the element.
- `getSize()`: This will return a promise with the size of the element. The size comes as a JSON object with `width` and `height` properties that indicate the height and width in pixels of the bounding box of the element. The bounding box includes padding, margin, and border.
- `getLocation()`: This will return a promise with the position of the element. The position comes as a JSON object with `x` and `y` properties that indicate the coordinates in pixels of the element relative to the page.
- `getAttribute(name)`: This will return a promise with the value of the specified attribute. Note that WebDriver does not distinguish between attributes and properties! If there is neither an attribute nor a property with that name, the promise will be fulfilled as `null`. If the attribute is a "boolean" HTML attribute (such as `checked` or `disabled`), the promise will be evaluated as `true` only if the attribute is present. If there is both an attribute and a property with the same name, the attribute value will be used.



If you really need to be precise about getting an attribute or a property, it is much better to use an injected script to get it.

- `getCssValue(cssPropertyName)`: This will return a promise with a string that represents the computed value of the specified CSS property. The computed value is the resulting value after the browser has applied all the CSS rules and the `style` and `class` attributes. Note that the specific representation of the value depends on the browser; for example, the `color` property can be returned as `red`, `#ff0000`, or `rgb(255, 0, 0)` depending on the browser. This is not cross-browser, so we should avoid this method in our tests.

- `findElement(locator)` and `findElements(locator)`: These will return an element, or all the elements that are the descendants of this element, and match the locator.
- `isElementPresent(locator)`: This will return a promise with a Boolean that indicates whether there is at least one descendant element that matches this locator.

As you can see, the `WebElement` API is pretty simple and allows us to do most of our tests easily. However, what if we need to perform some complex interaction with the UI, such as drag-and-drop?

Complex UI interaction

`WebDriverJS` allows us to define a complex action gesture in an easy way using the DSL defined in the `webdriver.ActionSequence` object. This DSL allows us to define any sequence of browser events using the builder pattern. For example, to simulate a drag-and-drop gesture, proceed with the following code:

```
var beverageElement = driver.findElement({ id: 'espresso' });
var orderElement = driver.findElement({ id: 'order' });

driver.actions()
  .mouseMove(beverageElement)
  .mouseDown()
  .mouseMove(orderElement)
  .mouseUp()
  .perform();
```

We want to drag an espresso to our order, so we move the mouse to the center of the espresso and press the mouse. Then, we move the mouse, by dragging the element, over the order. Finally, we release the mouse button to drop the espresso.

We can add as many actions we want, but the sequence of events will not be executed until we call the `perform` method. The `perform` method will return a promise that will be fulfilled when the full sequence is finished.

The `webdriver.ActionSequence` object has the following methods:

- `sendKeys(keys...)`: This sends a sequence of key events, exactly as we saw earlier, to the method with the same name in the case of `WebElement`. The difference is that the keys will be sent to the document instead of a specific element.

- `keyUp(key)` and `keyDown(key)`: These send the `keyUp` and `keyDown` events. Note that these methods only admit the modifier keys: *Alt*, *Ctrl*, *Shift*, *command*, and *meta*.
- `mouseMove(targetLocation, optionalOffset)`: This will move the mouse from the current location to the target location. The location can be defined either as a `WebElement` or as page-relative coordinates in pixels, using a JSON object with `x` and `y` properties. If we provide the target location as a `WebElement`, the mouse will be moved to the center of the element. In this case, we can override this behavior by supplying an extra optional parameter indicating an offset relative to the top-left corner of the element. This could be needed in the case that the center of the element cannot receive events.
- `mouseDown()`, `click()`, `doubleClick()`, and `mouseUp()`: These will issue the corresponding mouse events. All of these methods can receive zero, one, or two parameters. Let's see what they mean with the following examples:

```
var Button = require('selenium-webdriver').Button;

// to emit the event in the center of the espresso element
driver.actions().mouseDown(expresso).perform();
// to make a right click in the current position
driver.actions().click(Button.RIGHT).perform();
// Middle click in the espresso element
driver.actions().click(expresso, Button.MIDDLE).perform();
```



The `webdriver.Button` object defines the three possible buttons of a mouse: `LEFT`, `RIGHT`, and `MIDDLE`. However, note that `mouseDown()` and `mouseUp()` only support the `LEFT` button!

- `dragAndDrop(element, location)`: This is a shortcut to performing a drag-and-drop of the specified element to the specified location. Again, the location can be `WebElement` of a page-relative coordinate.

Injecting scripts

We can use WebDriver to execute scripts in the browser and then wait for its results. There are two methods for this: `executeScript` and `executeAsyncScript`.

Both methods receive a script and an optional list of parameters and send the script and the parameters to the browser to be executed. They return a promise that will be fulfilled with the result of the script; it will be rejected if the script failed.

An important detail is how the script and its parameters are sent to the browser. For this, they need to be serialized and sent through the network. Once there, they will be deserialized, and the script will be executed inside an autoexecuted function that will receive the parameters as arguments. As a result of this, our scripts cannot access any variable in our tests, unless they are explicitly sent as parameters. The script is executed in the browser with the `window` object as its execution context (the value of `this`).

When passing parameters, we need to take into consideration the kind of data that WebDriver can serialize. This data includes the following:

- Booleans, strings, and numbers.
- The `null` and `undefined` values. However, note that `undefined` will be translated as `null`.
- Any function will be transformed to a string that contains only its body.
- A `WebElement` object will be received as a DOM element. So, it will not have the methods of `WebElement` but the standard DOM method instead. Conversely, if the script results in a DOM element, it will be received as `WebElement` in the test.
- Arrays and objects will be converted to arrays and objects whose elements and properties have been converted using the preceding rules.

With this in mind, we could, for example, retrieve the identifier of an element, such as the following one:

```
var elementSelector = ".order ul > li";
driver.executeScript(
    "return document.querySelector(arguments[0]).id;",
    elementSelector
).then(function(id) {
    expect(id).to.be.equal('order_item0');
});
```

Notice that the script is specified as a string with the code. This can be a bit awkward, so there is an alternative available:

```
var elementSelector = ".order ul > li";
driver.executeScript(function() {
    var selector = arguments[0];
    return document.querySelector(selector).id;
}, elementSelector).then(function(id) {
    expect(id).to.be.equal('order_item0');
});
```

WebDriver will just convert the body of the function to a string and send it to the browser. Since the script is executed in the browser, we cannot access the `elementSelector` variable, and we need to access it through parameters. Unfortunately, we are forced to retrieve the parameters using the `arguments` pseudoarray, because WebDriver have no way of knowing the name of each argument.

As its name suggest, `executeAsyncScript` allows us to execute an asynchronous script. In this case, the last argument provided to the script is always a callback that we need to call to signal that the script has finalized. The result of the script will be the first argument provided to that callback. If no argument or `undefined` is explicitly provided, then the result will be `null`. Note that this is not directly compatible with the Node.js callback convention and that any extra parameters passed to the callback will be ignored. There is no way to explicitly signal an error in an asynchronous way.

For example, if we want to return the value of an asynchronous DAO, then proceed with the following code:

```
driver.executeAsyncScript(function() {
  var cb = arguments[1],
      userId = arguments[0];
  window.userDAO.findById(userId).then(cb, cb);
}, 'user1').then(function(userOrError) {
  expect(userOrError).to.be.equal(expectedUser);
});
```

Command control flows

All the commands in WebDriverJS are asynchronous and return a promise or `WebElement`. How do we execute an ordered sequence of commands? Well, using promises could be something like this:

```
return driver.findElement({name:'quantity'}).sendKeys('23')
  .then(function() {
    return driver.findElement({name:'add'}).click();
  })
  .then(function() {
    return driver.findElement({css:firstItemSel}).getText();
  })
  .then(function(quantity) {
    expect(quantity).to.be.equal('23');
  });
}
```

This works because we wait for each command to finish before issuing the next command. However, it is a bit verbose. Fortunately, with WebDriverJS we can do the following:

```
driver.findElement({name:'quantity'}).sendKeys('23');
driver.findElement({name:'add'}).click();
return expect(driver.findElement({css:firstItemSel}).getText())
  .to.eventually.be.equal('23');
```

How can the preceding code work? Because whenever we tell WebDriverJS to do something, it simply schedules the requested command in a queue-like structure called the control flow. The point is that each command will not be executed until it reaches the top of the queue. This way, we do not need to explicitly wait for the `sendKeys` command to be completed before executing the `click` command. The `sendKeys` command is scheduled in the control flow before `click`, so the latter one will not be executed until `sendKeys` is done.

All the commands are scheduled against the same control flow queue that is associated with the `WebDriver` object. However, we can optionally create several control flows if we want to execute commands in parallel:

```
var flow1 = webdriver.promise.createFlow(function() {
  var driver = new webdriver.Builder().build();

  // do something with driver here
});

var flow2 = webdriver.promise.createFlow(function() {
  var driver = new webdriver.Builder().build();

  // do something with driver here
});
webdriver.promise.fullyResolved([flow1, flow2]).then(function() {
  // Wait for flow1 and flow2 to finish and do something
});
```

We need to create each control flow instance manually and, inside each flow, create a separate `WebDriver` instance. The commands in both flows will be executed in parallel, and we can wait for both of them to be finalized to do something else using `fullyResolved`. In fact, we can even nest flows if needed to create a custom parallel command-execution graph.

Taking screenshots

Sometimes, it is useful to take some screenshots of the current screen for debugging purposes. This can be done with the `takeScreenshot()` method. This method will return a promise that will be fulfilled with a string that contains a base-64 encoded PNG. It is our responsibility to save this string as a PNG file. The following snippet of code will do the trick:

```
driver.takeScreenshot()
  .then(function(shot) {
    fs.writeFileSync(filePath, shot, 'base64');
  });
}
```



Note that not all browsers support this capability. Read the documentation for the specific browser adapter to see if it is available.



Working with several tabs and frames

WebDriver allows us to control several tabs, or windows, for the same browser. This can be useful if we want to test several pages in parallel or if our test needs to assert or manipulate things in several frames at the same time. This can be done with the `switchTo()` method that will return a `webdriver.WebDriver.TargetLocator` object. This object allows us to change the target of our commands to a specific frame or window. It has the following three main methods:

- `frame(nameOrIndex)`: This will switch to a frame with the specified name or index. It will return a promise that is fulfilled when the focus has been changed to the specified frame. If we specify the frame with a number, this will be interpreted as a zero-based index in the `window.frames` array.
- `window(windowName)`: This will switch focus to the window named as specified. The returned promise will be fulfilled when it is done.
- `alert()`: This will switch the focus to the active alert window.



We can dismiss an alert with `driver.switchTo().alert().dismiss();`.



The promise returned by these methods will be rejected if the specified window, frame, or alert window is not found.



To make tests on several tabs at the same time, we must ensure that they do not share any kind of state, or interfere with each other through cookies, local storage, or an other kind of mechanism.

Testing a rich Internet application

Now that we have a basic understanding of the capabilities of WebDriverJS and as we have clarified that we intend to test only our passive view in integration with the browser, we can start by setting up a project in the usual way: create a folder with the `lib/` and `test/` subfolders, issue the `npm init` command, and follow the instructions.

For the testing, we will use Mocha. Why not Cucumber.js? Well, it is a good idea to use Cucumber.js to test the core UI logic layer, as we will see later. However, passive view is a very technical layer in nature, and the only stakeholders that could be really interested in it would be the UX designer and the HTML/CSS expert. It is not uncommon that in some agile teams one or even two of these roles are fulfilled by a member of the team and not an external person. So, there is not much sense in adding the extra cost of using Gherkin.



Since tests that interact with browsers are usually slow, I recommend that you increase the timeout of the Mocha test to around 5 seconds. I have changed my test command inside the `package.json` file to: `mocha -u bdd -R spec -t 5000 -recursive`.

After executing the `npm init` command, we need to install the dependencies:

```
$ ~/mycafe/ui> npm install --save-dev browserify reactify chai chai-as-promised sinon sinon-chai mocha selenium-webdriver express
$ ~/mycafe/ui> npm install --save react
```

We have installed the `browserify`, `reactify`, and `selenium-webdriver` packages as development libraries. The first two packages will help us bundle the code of our passive view in order to be served to the test HTML page. The last one is the package that contains WebDriverJS.



I will develop the passive view using ReactJS (<http://facebook.github.io/react/>). That is why I am installing the react and reactify packages. Obviously, you can use your favorite framework for this, such as AngularJS (<https://angularjs.org/>) or Knockout (<http://knockoutjs.com/>). All the techniques in this chapter can be applied to them.

There is only one thing we need to get started; download the chromedriver file. This file contains the adapter of WebDriverJS for Google Chrome, so we need to download it. Just go to <http://chromedriver.storage.googleapis.com/index.html>, enter the folder for the latest release (at the time of writing, it was 2.13), download the correct ZIP for your operating system, and unzip it. The resulting executable file is the adapter. You can do one of the following:

- Put this file into the root of your project
- Create a symbolic link in the root of your project to a well-known standard location of the chromedriver executable on your machine
- Configure your PATH environment variable to include this file

Now, we are ready to begin coding!

The setup

First of all, we need to do a setup that is more complex than usual. We need to execute the code of our passive view inside one HTML page running in a browser. For this, we need to create the appropriate HTML. We need a web server not only to provide this page, but also to serve all the needed scripts. This includes bundling the code for our passive view in a way that can be consumed by a browser. Finally, we need to start a WebDriver session.

The test HTML page

There is no need for this page to be a real application page, just a container to include all the markup we need for the testing. In our example, we can create a `test/order.html` page like this:

```
<!DOCTYPE html>
<html>
<head lang="en">
    <meta charset="UTF-8">
```

```
<title>A test bed page for our Order Passive view</title>
<link href="/static/css/order.css" type="text/css" rel="stylesheet">
</head>
<body>
  <!-- To be used by injected scripts -->
  <script src="/node_modules/sinon/pkg/sinon.js"></script>
  <script src="/node_modules/chai/chai.js"></script>
  <script src="/node_modules/sinon-chai/lib/sinon-chai.js"></script>
  <script>window.expect = chai.expect;</script>
  <!-- To be used by common js bundles -->
  <script src="/dist/react.js"></script>
  <script src="/dist/order-view.js"></script>
  <!-- Some markup needed in our test-->
  <div class="not-a-container"></div>
  <div class="container"></div>
  <div class="not-a-container-either"></div>
</body>
</html>
```

It's just a simple page that includes some scripts and has some test markup. We only need some `<div>` elements to check that passive view is rendered inside the correct container. Besides this, we loaded the following scripts here:

- The `sinon`, `chai`, and `sinon-chai` packages. We will need them to perform some assertions against test doubles in our tests. Fortunately, the corresponding packages contain not only the package for Node.js, but also a version ready to be executed in the browser. Note that all the paths start with `/node_modules/<pkg_name>/` to be able to reach inside the corresponding package.
- The `/dist/react.js` file will contain the runtime of ReactJS. It will be served by `browserify`, so it can be used by our passive view package. We will see the details later.
- The `/dist/order-view.js` script will contain the *browserified* version of the passive view for the order.

Afterwards, we included a small snippet to globally expose on the page the `chai.expect` function for a commodity.

Serving the HTML page and scripts

Now we need to start a web server that will serve all of these resources. Let's create a `test/index.js` file with the following code:

```
'use strict';

var express = require('express'),
    port = process.env.PORT || 3000,
    server,
    app = express();

before('start web server', function (cb) {
    app.use(express.static(__dirname + '/../'));

    app.listen(port, function (err) {
        server = this;
        cb.apply(this, arguments);
    });
});

before('attach test utils', function () {
    this.uriForPage = function (name) {
        return 'http://localhost:' + port + '/test/' + name + '.html';
    };
});

after('stop web server', function (cb) {
    if (!server)
        return cb();
    server.close(function () {
        server = null;
        cb();
    });
});
```

This will start a static web server before the tests and stop it afterwards, there is nothing new here, since we use the same techniques as we did in the previous chapter. The only exception is that we attach a utility function, `uriForPage`, that will return the correct URI for a specified test page.

Using browserify to pack our code

This setup is OK to serve all the static assets we need, but the passive view code is written according to the CommonJS module convention, which is the one used by Node.js. The thing is that we can use `require()` to get other modules from our `order-view` module, and this will not work as it is in the browser. To solve this, some tools were invented; here, we will use the `browserify` package.

The `browserify` package will scan your modules to discover the tree of dependencies it needs. Then it will wrap each module in a wrapper function that implements the module pattern, so your code will behave as if it is in a Node.js-isolated module. It will also bundle the most basic Node.js modules, such as `events`, `utils`, `http`, and so on. Finally, it will pack all the code in a single file called `bundle` that can be consumed by a browser. The point here is that you do not need to code your UI logic in a different way just because it needs to be executed in a browser. You can develop it as if it were in Node.js and then, using `browserify`, the code will be bundled in a browser-friendly way.



The `browserify` tool has many options and is very powerful. It can be used as an API from JavaScript code or as a standalone command-line tool. It can be integrated easily with the most popular build systems in the JavaScript ecosystem, such as Grunt (<http://gruntjs.com/>) or Gulp (<http://gulpjs.com/>). To get all the details, visit <http://browserify.org/>.

Since I am using ReactJS, I am writing the passive view in JSX. This is a small extension to the JavaScript language that allows you to mix HTML and JavaScript. Just for illustration purposes, this is how my passive view looks after a few iterations of the TDD cycle:

```
/** @jsx React.DOM */

'use strict';

var React = require('react'),
    OrderView = require('./components/order.jsx');

function NOOP() {
}

module.exports = function (containerSelector, controller) {
    var onItemSelected = NOOP;
    if (controller)
```

Testing a UI Using WebDriverJS

```
onItemSelected = controller.itemSelected.bind(controller);

var view = React.renderComponent(
  <OrderView onItemClicked={onItemSelected}/>,
  document.querySelector(containerSelector)
);

return {
  update: view.setProps.bind(view)
};
};
```

You do not need to understand the file, but just notice that we are importing the react runtime using normal Node.js syntax. If we require additional files, then browserify will take care of this in a way that is transparent to us.



You do not need to write the passive view using JSX. In React, you can use plain JavaScript, so the reactify step can be skipped in this case. However, since using JSX is the common approach for most React users, I prefer to make a realistic example here. Obviously, this is not needed if you use AngularJS or Knockout.



We can modify our test/index.js file in the following way:

```
var express = require('express'),
  port = process.env.PORT || 3000,
  server,
  app = express(),
  browserify = require('browserify'),
  reactify = require('reactify'),
  bundles = {};

function registerBundle(name, cb) {
  return function (err, buf) {
    if (err)
      return cb(err);
    bundles[name] = buf.toString();
    cb();
  };
}
```

```
before('pack react', function (cb) {
  var reactFileName = require.resolve('react/dist/react.js');
  browserify({
    noParse: [reactFileName]
  })
    .require(reactFileName, {expose: 'react'})
    .bundle(registerBundle('react', cb));
});

before('pack order-view', function (cb) {
  var viewFileName = require.resolve('../lib/order-view.jsx');

  browserify()
    .transform(reactify)
    .require(viewFileName, {expose: 'order-view'})
    .add(viewFileName)
    .exclude('react')
    .bundle(registerBundle('order-view', cb));
});
```

Just before we started the tests, we used `browserify` to process the `react` runtime and our passive view code. The resulting bundles are stored in memory in the `bundles` object. Without entering into too much detail about `browserify`, we can imagine how all of this works. On one side, the `react` package includes an already bundled file that is ready to be used in the browser, so there is no point in applying `browserify` again to it. On the other hand, the `react` module needs to be visible somehow to our `order-view` module because it imports the `react` module. Here is the solution:

- In the '`pack react`' block, the `react` distribution is simply wrapped around a `browserify` wrapper and exposed as a package named `react`. We explicitly told `browserify` not to try to analyze the code of the file, thus saving time.
- In the '`pack order-view`' block, we told `browserify` to pack the `order-view` module and all its dependencies in a single file, but we explicitly excluded the `react` dependency because we will serve it in a separate bundle.

This setup is an efficient way to bundle our code, since it avoids parsing and analyzing the whole `react` runtime again and again. This is important in a real project because you will probably want to test several views and not only the one for the orders. Actually, in a real project, you would probably end up sharing some of this code between the setup of the WebDriver tests and the build system.

We just need to add some routes to our web server to serve the bundles:

```
before('start web server', function (cb) {
  app.use(express.static(__dirname + '/../'));

  app.get('/dist/:bundleName.js', function (req, res) {
    var bundle = bundles[req.param('bundleName')];
    if (!bundle)
      return res.sendStatus(404);
    res.set('Content-Type', 'application/json');
    res.send(bundle);
  });

  app.listen(port, function (err) {
    server = this;
    cb.apply(this, arguments);
  });
});
```

The code just looks in the dictionary and returns the bundle as JSON if found. If not, it will return a 404 error.

Maybe you are wondering why we executed `browserify` in the `before()` block instead of in the build pipeline, using either Gulp or Grunt. We could make the test task dependent on the distribution task. After all, you need to do it anyway to correctly build a distribution of your UI. In fact, this approach is perfectly fine, but I slightly prefer the one used in this chapter. I have some reasons for this:

- I do not want to make the build pipeline more complex than necessary.
- I want the test suite to be as fast as possible, so I am only packing the minimum set of JavaScript files that the test page needs. Note that, in the test page, we only test one single passive view. Probably, in our distribution, we will build bigger bundles because a real page can use several widgets.
- If you need to make a change in the setup, you only need to modify the test setup. With the other approach, you need to modify the build pipeline too.

Anyway, both approaches have their advantages, so if you prefer to add a bit more complexity to your build pipeline and make your test code a bit simpler, then try it!

Creating a WebDriver session

The only thing we need now is to create a WebDriver session. When a session is created, a new browser is started and put under the control of a WebDriver instance. There are multiple ways of creating a session, but we will go with the most straightforward one: using chromedriver to control a single Google Chrome browser instance. For this, we need to add the following lines of code to our `test/index.js` file:

```
var webdriver = require('selenium-webdriver');
before('start web driver session', function () {
  this.driver = new webdriver.Builder().
    withCapabilities(webdriver.Capabilities.chrome()) .
    build();
});

after('quit web driver session', function () {
  return this.driver.quit();
});
```

The code is quite simple. We used the `webdriver.Builder` object to create a WebDriver instance that, under the hood, will open a browser session. The `withCapabilities` method specifies which kind of browser we want; in this case, we want a Google Chrome browser (`webdriver.Capabilities.chrome()`).

The `after()` block will simply destroy the session when all the tests are done.



You can go to http://selenium.googlecode.com/git/docs/api/javascript/class_webdriver_Capabilities.html to know which browsers are currently supported.



Testing whether our view updates the HTML

We are now ready to start writing a test for our view. The first feature that a passive view should have is the ability to receive a view model and update the HTML with the new information. A view model is a simple JSON object with all the information needed by the view to update the DOM. It not only includes the information to show, but also includes the state of the controls, whether some element should be highlighted or hidden, and so on.

The general pattern to test this kind of feature is as follows:

1. Set up your test by executing a remote script where you instantiate an instance of your passive view and attach it to the corresponding HTML tag.
2. Drive your test with a remote script that will invoke the passive view update method with an example of a view model.
3. Check whether the relevant HTML nodes have been created or updated using WebDriver.
4. Triangulate with several examples of the view model.

Let's see how all of this can be done with some code. Create a `test/order_view_updates_dom.js` file with the following lines of code:

```
'use strict';

var chai = require('chai'),
    expect = chai.expect;

chai.use(require('chai-as-promised'));

var driver;
before(function () {
  driver = this.driver;
});

describe('An order-view updates the DOM', function () {
  before(function () {
    driver.get(this.uriForPage('order'));

    return driver.executeScript(function () {
      window.view = require('order-view')('.container')
    });
  });

  function willUpdateTheDOM(example) {
    // Add tests here!
  }

  [
    {
      ...
    }
  ].forEach(willUpdateTheDOM);
});
```

```
        description: 'an order',
        viewModel: {
            totalPrice: "12.34 $"
        }
    }
].forEach(willUpdateTheDOM);
});
```

Note how in the setup we are instructing the browser to load the order HTML test page. For this, we used the WebDriver `get()` method. Then we simply used `executeScript` to create a new view instance in the `<div>` with the `.container` class. We stored the resulting instance in `window.view` for future reference. Remember that actually this code is being executed in the browser, so we have access to the `window` object.

There is an interesting trick here; we are using `before()` instead of `beforeEach()`, so we can load the HTML page, and create the view instance, only once for all the tests, instead of one time per each test. This will speed up our test suite, but it has the risk of mixing the state between the tests. Fortunately, the passive view is a stateless object, so it is safe to do it like this. At most, you would need to reset the HTML and/or other browser states in an `afterEach()` block, as follows:

```
afterEach(function() {
    return driver.executeScript(function () {
        document.querySelector('.container').innerHTML = '';
    });
});
```

Fortunately this is not necessary with ReactJS, since the HTML will be fully updated whenever we update the view, removing the old HTML if necessary.

Inside the `willUpdateTheDOM` function, we will code the parameterized tests we need. This function will be invoked with several examples of the view model to triangulate the test. For now, we only have a very simple example showing the `totalPrice`. Note that this field is a correctly formatted and localized string. The format and localization logic is implemented in the core UI layer and tested independently, so we do not need to worry about the values here. Just use meaningful examples and check whether the data is displayed as it is in the view model.

Let's write some test code to check whether `totalPrice` is displayed. Add the following lines of code inside the `willUpdateTheDOM` function:

```
function willUpdateTheDOM(example) {
    var viewModel = example.viewModel;

    describe('when update is called with ' + example.description,
    function () {
        beforeEach(function () {
            return driver.executeAsyncScript(function () {
                var viewModel = arguments[0],
                    cb = arguments[1];

                view.update(viewModel, cb);
            }, viewModel);
        });
    });

    it('will update the DOM to show the total price', function () {
        var priceElement = driver.findElement({
            css: '.container .order .price'
        });

        return expect(priceElement.getText())
            .to.eventually.be.equal(viewModel.totalPrice);
    });
}
}
```

We executed the action we want to test using a remote asynchronous script. We just executed `view.update` with the corresponding view model and a callback. The `view.update` operation has been designed as asynchronous because some frameworks, such as React, perform an incremental update of the DOM in asynchronous batches for performance reasons.

The assertion simply uses WebDriver to locate the element that shows the price with a CSS selector. We called `getText()` on the resulting element that will give us a promise with the text shown earlier. We can easily assert that this text is the same as the one in the view model, thanks to the `sinon-as-promised` plugin of Chai.

We can start incrementally adding tests to check whether more information is shown. For example, we could test whether the order items are shown as rows in a table:

```
function willUpdateTheDOM(example) {
  var viewModel = example.viewModel;

  describe('when update is called with ' + example.description,
  function () {
    // Skipped for brevity

    describe('will update the DOM to show the items', function () {
      it('there is one entry per each item', function () {
        var itemElements = driver.findElements({
          css: '.container .order .item'
        });

        return expect(itemElements).to.eventually
          .have.length(viewModel.items.length);
      });

      viewModel.items.forEach(function (itemModel, i) {
        var itemSelector = '.container .order .item:nth-of-type(' + (i
          + 1) + ')';

        it('the DOM for item ' + i + ' shows the item name', function
        () {
          var itemNameElement = driver.findElement({
            css: itemSelector + ' .name'
          });

          return expect(itemNameElement.getText())
            .to.eventually.be.equal(itemModel.name);
        });

        it('the DOM for item ' + i + ' shows the item quantity',
        function () {
          var itemQuantityElement = driver.findElement({
            css: itemSelector + ' .quantity'
          });

          return expect(itemQuantityElement.getText())
            .to.eventually.be.equal(itemModel.quantity);
        });
      });
    });
  });
}
```

```
});

it('the DOM for item ' + i + ' shows the item price', function()
{
    var itemPriceElement = driver.findElement({
        css: itemSelector + '.price'
    });

    return expect(itemPriceElement.getText())
        .to.eventually.be.equal(itemModel.unitPrice);
});

});

});

}

[
{
    description: 'an order with 3 items',
    viewModel: {
        totalPrice: '12.34 $',
        items: [
            {name: 'Expresso', quantity: '2', unitPrice: '2.33 $'},
            {name: 'Mocaccino', quantity: '3', unitPrice: '1.45 $'},
            {name: 'Latte', quantity: '1', unitPrice: '2.00 $'}
        ]
    }
}
].forEach(willUpdateTheDOM);
```

We just added a loop to create a test for each item that will check the price, quantity, and name. Here, note that the use of the `nth-of-type(index)` selector allows us to locate individual item entries.

We should also test how the user actions are rendered. How do we represent a user action in the view model? Let's have a look at this:

```
[

{
    description: 'an order with 3 items',
    viewModel: {
        totalPrice: '12.34 $',
        items: [
```

```

        {name: 'Expresso', quantity: '2', unitPrice: '2.33 $'},
        {name: 'Mocaccino', quantity: '3', unitPrice: '1.45 $'},
        {name: 'Latte', quantity: '1', unitPrice: '2.00 $'}
    ],
    addBeverageForm: {
        target: '/orders/items_3',
        method: 'POST',
        enabled: true,
        shown: true,
        fields: [
            {name: '__method', type: 'hidden', value: 'PUT'},
            {name: 'beverage', type: 'text', value: '', error: true},
            {name: 'quantity', type: 'text', value: '1'},
            {name: 'addToOrder', type: 'button',
                value: 'Add to order'}
        ],
        messages: ['name of the beverage is required']
    }
}
]
.forEach(willUpdateTheDOM);

```

As shown, a good way to represent a user action is using a form. In the view model, we have opted to model the form in a low-level way. It is an object that contains several properties, explained here:

- The `enabled` and `shown` properties define whether the form's controls are enabled or not or whether it is going to be visible or not.
- The `fields` array contains an object for each field. Each one of these objects represents an input control, so we need properties to define the name, the type of the field (`hidden`, `text`, `submit`, `button`, `file`, and so on), the value, and whether the field is marked as containing an error or not.
- The `method` and `target` fields will define which HTTP verb will be used if we submit the form and where the URI is that executes the form submission.
- Finally, it contains a `messages` array that contains the possible list of error messages to be shown.

Now we need to test whether the form is rendered correctly:

```
function willUpdateTheDOM(example) {
    var viewModel = example.viewModel;
```

```
describe('when update is called with ' + example.description,
function () {
    // Skipped for brevity
    describe('will update the DOM to show the add beverage action',
function () {
    var formModel = viewModel.addBeverageForm;
    describe('there is a form', function () {
        beforeEach(function () {
            this.form = driver.findElement({
                css: '.container .order form.add-beverage'
            );
        });

        it('with a ' + formModel.method + ' method', function () {
            return expect(this.form.getAttribute('method'))
                .to.eventually
                .be.equal(formModel.method.toLowerCase());
        });

        it('with action set to ' + formModel.target, function () {
            return expect(this.form.getAttribute('action'))
                .to.eventually
                .match(new RegExp(formModel.target + '$'));
        });
        // Skipped for brevity
    });
});
});
```

The first thing we did is to look for the form using a CSS selector and save it in the `this.form` variable. This way, we can remove the duplicated code that looks for the form in the different tests. In the preceding code, we checked the `method` and `action` properties of the form. There is no complicated code in this test. There are just two caveats:

- The `method` property of a form must be in lowercase, as specified in the HTML standards.
- The browser will return the full URI as the `action` property value. That is why we are asserting using a regular expression.

We can also check whether the form is shown or hidden and whether the messages are shown:

```
describe('will update the DOM to show the add beverage action',
  function () {
    // Skipped for brevity
    it('which is ' + (formModel.shown ? '' : 'not ') + 'visible',
      function () {
        return expect(this.form.isDisplayed())
          .to.eventually.be.equal(formModel.shown);
      });

    if (formModel.shown) {
      formModel.messages.forEach(function (msg, i) {
        it('with an error message [' + msg + ']', function () {
          var msgElement = this.form.findElement({
            css: '.error-msg:nth-of-type(' + (i + 1) + ')'
          });

          return expect(msgElement.getText())
            .to.eventually.equal(msg);
        });
      });
    }
    // Skipped for brevity
  });
});
```

To assert whether the form is shown or not, we checked the `isDisplayed()` method from WebDriver. Also, note the way the title of the test is changed depending on whether the example says that the form should be shown or not.

If the form is not displayed, then the `getText()` message will return an empty string. That is why the test is executed conditionally, based on whether the form is displayed or not. Note how we locate the elements for each message using `this.form.findElement`. This way, we only need to use a CSS selector that is relative to the form. This small trick could have been used for the tests of the items we saw earlier, instead of using a long selector.

Finally, we need to check the fields:

```
describe('will update the DOM to show the add beverage action',
function () {
    // Skipped for brevity
    formModel.fields.forEach(function (fieldModel) {
        describe('with a field named ' + fieldModel.name, function () {
            beforeEach(function () {
                this.field = this.form.findElement({
                    css: 'input[name="' + fieldModel.name + '"]'
                });
            });

            it('that has type [' + fieldModel.type + ']', function () {
                return expect(this.field.getAttribute('type'))
                    .to.eventually.be.equal(fieldModel.type);
            });

            it('that has value [' + fieldModel.value + ']', function () {
                return expect(this.field.getAttribute('value'))
                    .to.eventually.be.equal(fieldModel.value);
            });

            it('that is ' + (fieldModel.error ? '' : 'not ') +
'highlighted as error', function () {
                var className = this.field.getAttribute('class');

                if (fieldModel.error)
                    return expect(className)
                        .to.eventually.include('error');
                else
                    return expect(className)
                        .to.eventually.not.to.include('error');
            });

            it('that is ' + (formModel.enabled ? 'enabled' : 'disabled'),
function () {
                var disabled = driver.executeScript(function () {
                    var inputEl = arguments[0];
                    return inputEl.disabled;
                }, this.field);
            });
        });
    });
});
```

```
        return expect(disabled).to.eventually
            .be.equal(!formModel.enabled);
    });
});
});
});
});
});
```

The tests are very simple and use the techniques we have seen until now. The only new things are the tests for `enabled` and `error`. The test for `error` accesses the `class` attribute and checks whether it contains the `.error` class or not. Note that we need to use an `if` statement here. The test is parameterized, so we need to change the assertion depending on whether the field should be marked as an error or not. If it is marked as an error, we use the `include('error')` Chai assertion to check whether the string containing the class names includes `'error'`. If the field is not marked as an error, we use `not.to.include('error')` to test exactly for this opposite.

The test for `enabled` is more interesting since we do not need to check the `disabled` Boolean attribute, just the JavaScript property. As we saw earlier, in this case we cannot use `getAttribute` since it gives priority to the attribute. That is why a remote script is used. We can directly pass the `WebElement` script, and this will be converted to a DOM element from which we can read the JavaScript `disabled` property.



Note the kind of CSS selectors we used. They are semantic and deliberately loose coupled with the specific structure of the resulting HTML. This results in tests that are less brittle to the changes in the HTML structure. The price we pay for this is that we do not check which specific HTML is generated. However, sometimes this detail is important; in these cases, we should use more specific selectors that reveal the detailed structure of the HTML. In general, we should aim for the less-specific selectors that check exactly what we want to.

We can continue adding elements to our view model and test whether they are shown appropriately. This give us an incremental workflow to test-drive the implementation of the UI.

Testing whether our view reacts with the user

There is another feature that we should test: the ability of the passive view to receive low-level DOM events and transform them into appropriate calls to the core UI logic. It is important to realize that mapping between DOM events and actions in the core UI logic is not one-to-one:

- Several low-level DOM events can be mapped to exactly the same action. For example, if we are using a search form, we can trigger the search both by pressing *Enter* or by clicking on a search button.
- A complex sequence of DOM events could be mapped to a single action. For example, a drag-and-drop action can be mapped to an "add to order" action.



It is a good idea to talk with the UX designer about how exactly the user accomplishes business operations in the UI and when validations should be triggered to discover the mapping between low-level DOM events and calls in the core UI layer.

The pattern to test this kind of features is as follows:

1. Set up a test double of your core logic UI with spies for each relevant call you want to check using a remote script.
2. Drive the test using WebDriver to send native events.
3. Use WebDriver to execute a remote script that checks the spies.



It is a good practice to create a separate scenario for each native event that maps to the same action and to separate test files containing all the scenarios that check the same logic action. This way, we can easily locate the relevant test suite file.

Let's see how all of this can be done with some code. Create a `test/order_view_fires_addBeverage.js` file where we will test which events perform an `add beverage` operation against the core UI logic:

```
describe('An order-view sends an "add beverage" request to the controller', function () {
  var addBeverageForm = {
    target: '/orders/items_2',
    method: 'POST',
    enabled: true,
    shown: true,
```

```

fields: [
  {name: '__method', type: 'hidden', value: 'PUT'},
  {name: 'beverage', type: 'text', value: ''},
  {name: 'quantity', type: 'text', value: ''},
  {name: 'addToOrder', type: 'submit', value: 'Add to order'}
],
messages: []
};

before(function () {
  return driver.get(this.uriForPage('order'));
});

beforeEach(function () {
  return driver.executeAsyncScript(function () {
    var newOrderView = require('order-view'),
      addBeverageForm = arguments[0],
      cb = arguments[1];

    window.controller = {
      addBeverage: sinon.spy()
    };

    newOrderView('.container', window.controller)
      .update({
        totalPrice: '0 $',
        items: [],
        addBeverageForm: addBeverageForm
      }, cb);
    addBeverageForm;
  });
});

```

The preceding code performs the setup necessary for our tests:

- We defined a variable called `addBeverageForm` that will hold the initial view model of the form we are testing.
- We loaded the test page in a `before` block, so it will be done only once.
- Finally, we used a `beforeEach` block to initialize our passive view using `executeAsyncScript`. This setup is a bit different from the one we did in the previous feature. As we did earlier, we created a new instance of the passive view and attached it to the `.container` element. However, this time we provided a test double for the controller that will receive the user operations. In this case, the controller will have an `addBeverage` method that should be called when the user executes the form. We finally updated the view using a view model that contains the `addBeverageForm` model.

Note that, unlike in the previous feature, we are using a `beforeEach` block instead of a `before` block to initialize our view. This is because we will change the value of the inputs of the forms when we tell WebDriver to type into them. If we do not regenerate the HTML again, the next test will type again in the input but the input has already been filled with some text. So our final input will consist of a concatenation of both texts: the one for the old test and the one for the new test. Another important reason for using a `beforeEach` block is that we need to create a new test double so the tests do not interfere with each other.

Another option would have been using a `before` block as we did in the previous feature, but also adding an extra `afterEach` block to do a clean up. It is a bit more complex, but we usually avoid the expensive operation of recreating the whole HTML and the passive view instance whenever we prepare a new test. If we went with this approach, we would end up with the following setup:

```
before(function () {
  driver.get(this.uriForPage('order'));

  return driver.executeAsyncScript(function () {
    var newOrderView = require('order-view'),
      addBeverageForm = arguments[0],
      cb = arguments[1];

    window.controller = {
      addBeverage: sinon.spy()
    };

    newOrderView('.container', window.controller)
      .update({
        totalPrice: '0 $',
        items: [],
        addBeverageForm: addBeverageForm
      }, cb);
    }, addBeverageForm);
  });

  afterEach(function () {
    driver.findElement({
      css: '.container .order form.add-beverage input[name="beverage"]'
    }).clear();
  });
}
```

```
driver.findElement({
  css: '.container .order form.add-beverage input[name="quantity"]'
}).clear();

return driver.executeScript(function () {
  controller.addBeverage.reset();
});

});
```

In the `afterEach` block, we found the input fields and cleared them; then we executed a remote script to reset the Sinon spy.

Deciding between the "extra `afterEach` method for clean up with a single `before` block" approach and the "beforeEach block with the page loading in a separate `before` block" approach depends mainly on which one makes your test faster and whether the clean up is complex or not.

Let's write a test that ensures that, when the user clicks on the `addToOrder` button in the form, then the controller receives an `addBeverage` request. We can add the following code after our setup:

```
describe('given that the user has entered 2 Cappuccinos', function () {
  var expectedRequest = {
    beverage: 'Cappuccino',
    quantity: '2',
    target: '/orders/items_2',
    method: 'PUT'
  };

  beforeEach(function () {
    driver.findElement({
      css: '.container .order form.add-beverage
input[name="beverage"]'
    }).sendKeys('Cappuccino');

    driver.findElement({
      css: '.container .order form.add-beverage
input[name="quantity"]'
    }).sendKeys('2');
  });
});
```

```
it('when the user clicks the "add to order" button, ' +
'an addBeverage request will be sent to the order with "2
Cappuccinos"', function () {
  driver.findElement({
    css: '.container .order form.add-beverage
input[name="addToOrder"]'
  }).click();

  return driver.executeScript(function () {
    expect(controller.addBeverage)
      .to.have.been.calledWith(arguments[0]);
  }, expectedRequest);
});
});
```

We used a `beforeEach` block to fill both input controls using the `sendKeys` method. We do not need to make them wait for each other; WebDriver will take care of this using its control flow mechanism. Note that the `beforeEach` block is not asynchronous, so it will finish as soon as we schedule both `sendKeys` commands, without waiting for them to be executed.

Now that we have filled the add beverage form, we can perform the test itself. A `click` event is sent to the `addToOrder` button, and then we send a remote script that will make an assertion on the controller test double. In this case, we expect that the `addBeverage` method has been called with the correct parameters, an object with a property with the value of each input field, and the correct target URI and HTTP methods. Note that we expect the passive view not to use the `method` parameter of the form, but the value of the `_method` hidden input instead.

This way of writing tests works because WebDriver will schedule `sendKeys`, `click`, and `executeScript` in the same order as we did in the code. It will wait for each command to finish before executing the next one, so we do not need to do anything special to orchestrate them. The only thing we need to remember is to return the promise of the `executeScript` command so that Mocha will wait for all the commands and for the assertion to be finished.

The first time you run this test, you will probably see it fail in a weird way. It will probably tell you that the `expect` symbol is undefined. What is happening is that, when we click on a `submit` control, the browser will execute the form and navigate outside our test page. WebDriver will wait for the `click` event to finish; this usually involves waiting for the submission to finish. Then it will execute the remote script but, in the new page, we have not declared any `chai` or `expect` dependency.

To be able to pass this test, the passive view needs to kidnap the submit event of the form and stop its default behavior: submitting the form and navigating out of the page. This can be done with a simple `ev.preventDefault()` method in the event handler of the form. This is a normal technique in single-page applications related to the progressive enhance approach.

Now we want to test whether the same `addBeverage` request is issued when the user presses `Enter` in any of the fields. Let's add a test for this:

```
describe('given that the user has entered 2 Cappuccinos', function () {
    // Skipped for brevity

    ['beverage', 'quantity'].forEach(function (fieldName) {
        it('when the user press ENTER in the "' + fieldName + '" input, ' +
            'an addBeverage request will be sent to the order with "2
            Capuccinos"', function () {
            driver.findElement({
                css: '.container .order form.add-beverage input[name="' +
                    fieldName + '"]'
            }).sendKeys(Key.ENTER);

            return driver.executeScript(function () {
                expect(controller.addBeverage)
                    .to.have.been.calledWith(arguments[0]);
            }, expectedRequest);
        });
    });
});
```

The test is similar to the one we did earlier. We just simply pressed the `Enter` key in the specified input field. The only thing here is that the test is parameterized against the input name, and we ran it for the `beverage` and `quantity` fields.

Now we can triangulate to check whether these fields are used in the `addBeverage` call if we pass different values for them:

```
function willSendAnAddBeverageRequest(example) {
    var enteredBeverage = example.input.beverage,
        enteredQuantity = example.input.quantity;
    describe('given that the user has entered ' + example.title,
        function () {
            var expectedRequest = {
                beverage: enteredBeverage,
```

```
        quantity: enteredQuantity,
        target: '/orders/items_2',
        method: 'PUT'
    };

    beforeEach(function () {
        driver.findElement({
            css: '.container .order form.add-beverage
input[name="beverage"]'
        }).sendKeys(enteredBeverage);

        driver.findElement({
            css: '.container .order form.add-beverage
input[name="quantity"]'
        }).sendKeys(enteredQuantity);
    });

    it('when the user clicks the "add to order" button, ' +
       'an addBeverage request will be sent to the order with "' +
example.title + '"', function () {
        // Skipped for brevity
    });

    ['beverage', 'quantity'].forEach(function (fieldName) {
        it('when the user press ENTER in the "' + fieldName + '" input,
' +
           'an addBeverage request will be sent to the order with "' +
example.title + '"', function () {
            // Skipped for brevity
        });
    });
});

[
{
    title: '2 Capuccinos',
    input: {
        beverage: 'Cappuccino',
        quantity: '2'
    }
},
{

```

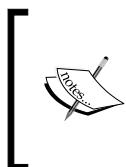
```

title: '12 Expressos',
input: {
  beverage: 'Expresso',
  quantity: '12'
},
{
  title: 'nothing',
  input: {
    beverage: ' ',
    quantity: ' '
  }
}
].forEach(willSendAnAddBeverageRequest);

```

The last scenario is interesting. Here, we test what happens when the user enters empty spaces in the fields. What should the passive view do? Nothing special! It should just send exactly the same spaces the user entered. The passive view is not responsible for the validation, but the UI control logic is. So passive view tests must not say anything about what is a correct input and what is not. Leave this to the tests of the UI control logic.

We can continue adding tests for any other DOM-level event that needs to be translated to an `addBeverage` request. Which ones? I don't know; the UX designer will tell you!



Sending any kind of event to the browser, as we did with the `click` and `sendKeys` methods, is very slow. Especially, the `sendKeys` method because it will send one event for each character and key that we pass as parameters. This makes this kind of test inherently slow, more than 200 milliseconds each.

What about our UI control logic?

Until now, we have tested only the interaction between the HTML and our passive view. However, this layer is dumb; it is really not doing much. We still have a lot of logic to test, and this logic is owned by the core UI layer.

The good news is that the logic in this layer is not very different from what we saw in *Chapter 3, Writing BDD Features*, and *Chapter 4, Cucumber.js and Gherkin*. The point is that we can use the same techniques to test it.

Who are our stakeholders here? Anyone who has an interest in how users interact with other users. In most organizations, this implies business people, regular/test users, UX designers, and so on. Often, these stakeholders are not really engineer people, so it makes sense, as we saw in *Chapter 4, Cucumber.js and Gherkin*, to write our BDD features for the UI using Gherkin. This way, the interaction with them will be better, and the status of the development will be easier for them to understand.

If we stick to a language that is meaningful to these stakeholders and relevant for the UI domain, we will end up with features that deal with a single interaction of the user with the UI and how these interactions are reflected in the UI.

For example, the most simple feature would be to define what happens when the user displays an order:

```
Scenario:  
  Given that the order contains the following items:  
    | beverage | quantity | price |  
    | Espresso | 3       | 1     |  
    | Capuccino| 2       | 5     |  
  And that the total price is "10 dollars"  
  When the user displays the order  
  Then "10.00 $" will be shown as total price  
  And "3.00 $" will be shown as a discount  
  And the following order items will be shown:  
    | item        | unit price | subtotal |  
    | 3 Expressos | 2.00 $    | 2.00 $   |  
    | 2 Capuccinos| 5.00 $    | 10.00 $  |
```

This is actually very similar to the feature we had in the core logic, but the implementation of the steps is slightly different. As a general setup, we need to create test doubles for the passive view and the server client and attach them to the core UI logic. To ensure that the order has certain contents, we just need to set up the test double for the server side. To drive the feature, we just call a `display` method in the core UI controller. To check whether the UI is updated, we need to check whether the passive view's `update` method has been called with the correct parameters.

In this Gherkin, we tackled aspects such as what kind of information is shown (the subtotal for each entry, the total price, and whether we got some discount). We also showed you how to format the information correctly: quantity and beverage name are together in the same element, and the amount of money is formatted to 2 decimal places and the use of the dollar symbol.

 Note that we do not say anything about how exactly the UI looks. This would be a mistake. The exact way in which the HTML should be updated is one responsibility of the passive view, and we should avoid this kind of detail here. Thus, steps such as "The second row of the order items table will show <the data of the item>" are a bad smell, and we should aim for something like "The <data of the item> will be shown in the second position". Here, we do not care about whether we show it with a table or not, but whether we show the correct information in the correct order.

Another example would be on validating a field in the client side without going to the server:

```
Scenario: quantity is incorrect
When the user fills "1Df" in the quantity field
Then the quantity field will be highlighted as erroneous
And the quantity field will show "1Df"
And an "invalid quantity (1Df)" message will appear
```

The steps are very simple to implement. We do not have a setup here, although we would need to have it if the validation depends on the value of the other fields. The "user fills" action corresponds to a method in the core UI logic controller that will be invoked by the passive view when appropriate. The assertion again is simply checking the parameters passed to the update method in the passive view.

Again, sticking to the correct level of abstraction here is very important. The following details are all irrelevant for this layer and should be dealt with in the passive view:

- How does the user enter the data in the field? Maybe they press *Enter* or change the focus to another field? Or maybe, they just expect to update the field as they type?
- What exactly is a field? A fancy rich control? A simple `<input>` tag? Or maybe an editable `<div>`?
- How does the field get highlighted? A red star appears next to it? Its border changes color?
- How does the messages exactly look? Where are they positioned?

This kind of detail should not be expressed in this Gherkin.

Suppose we want to specify when the quantity of the item is filled successfully:

```
Scenario: quantity is correct
  Given that the user is Spanish
  When the user fills " 1000 " in the quantity field
  Then the quantity field will show "1.000"
  And the "add beverage" action will be enabled
```

This scenario shows that we accepted blanks and that they are stripped. It also shows that, because the user is from Spain, the quantity is formatted with the "." character as a thousands separator.

If you keep the right abstraction level, your BDD features will be very robust and will not change whenever the graphic design changes or there are small changes in the interactivity. The features should only be changed if there is a big change in the way the UI behaves—changes in the navigation; changes relating to when controls are disabled or enabled, or when they appear or disappear; and changes in client-side validation, formatting, and so on.

Summary

This chapter showed us that a good way to test the UI of an application is actually to split it into two parts and test them separately. One part is the core logic of the UI that takes responsibility for control logic, models, calls to the server, validations, and so on. This part can be tested in a classic way, using BDD, and mocking the server access. No new techniques are needed for this, and the tests will be fast. Here, we can involve nonengineer stakeholders, such as UX designers, users, and so on, to write some nice BDD features using Gherkin and Cucumber.js.

The other part is a thin view layer that follows a passive view design. It only updates the HTML when it is asked for, and listens to DOM events to transform them as requests to the core logic UI layer. This layer has no internal state or control rules; it simply transforms data and manipulates the DOM. We can use WebDriverJS to test the view.

This is a good approach because the most complex part of the UI can be fully test-driven easily, and the hard and slow parts to test the view do not need many tests since they are very simple. In this sense, the passive view should not have a state; it should only act as a proxy of the DOM.

The setup is more complex than usual because we need to bundle the code we want to test in a way it can be consumed by the browser. We can use the `browserify` tool for this. We can execute `browserify` during the setup or as part of our build process. There is no need to use a special package system just because we are coding for the browser.

As part of the setup, we will often need to inject test doubles for our passive view using the `executeScript` method of `WebDriverJS`. We do not need to always reload the test page between tests. Since the view is stateless, we can just reset the test double for the core UI logic and wipe out the generated HTML from the last test. This will speed up the tests. For additional cleanup, we can clean up cookies and other resources using `WebDriver`.

To drive our UI during the tests, we should send the same gestures that the user would do to interact with the UI. We can do this with various methods in the `elements` returned by `WebDriver`; alternatively, for more complex interactions, we can use the `ActionSequence` object.

In our assertions, we can use `executeScript` to check whether the passive view requested the correct operation to the core UI logic. If the expectation was not met, the assertion error will be transferred to our test.

In the assertions, we also need to check whether the HTML is updated correctly. For this, we can use `WebDriver` again. We can find the HTML elements using the `findElement` or `findElements` methods. To locate the elements, I recommend that you use CSS selectors, although other locators are available.

Regardless of all these techniques, UI tests are slow and inherently brittle to the changes in the HTML structure. To complicate things more, the layer that changes more frequently is the passive view, not the core UI layer. Can we do something about it? Well, there are no silver bullets here, but we can make it a bit better with the Page Object pattern, as we will see in the next chapter.

7

The Page Object Pattern

In this chapter, we will try to better organize the test codebase we created in the previous chapter. In order to do so, you will learn the Page Object pattern. This pattern will allow us to simplify our test code and encapsulate the complexity of using WebDriver and accessing the DOM in a nice object that we can reuse across all of our codebase.

In this chapter, you will learn:

- What the Page Object pattern is and why to use it
- Best practices to design a page object
- How to save code by designing a library of small reusable page objects
- How to build a page object using WebDriver
- How to properly test the navigation logic of our application (do not do this in the page object)

Introducing the Page Object pattern

If we have a look at the code we have written, we will realize that the tests are a bit ugly. The tests are not very readable because there is a lot of references to the WebDriver API that is a bit low-level. Furthermore, there are many CSS selectors throughout all the tests. This produces two main problems:

- Excessive verbosity and a lack of readability in the tests due to all these references to CSS selectors and the WebDriver API.

- Difficult maintenance. A change in the structure of the HTML can break our tests, because the specified CSS selectors are no longer valid. The problem is not the fact that the tests get broken, but that we need to review all the tests, looking for the selectors that we need to fix. This is because the same CSS selector can be referenced in several tests that depend on the same element.

The Page Object pattern intends to solve these problems by encapsulating the details of accessing the elements of a page and interacting with it. The idea is to create an object that offers a logical view of the page and move all the references to the WebDriver and CSS selectors inside this object.

Besides this, the Page Object pattern has an additional advantage: it decouples our test from WebDriver so that we can switch to another tool if we want to. After all, technology moves fast, and in a year or two there could be better tools than WebDriver out there. In this event, it would be nice to change to such a tool without modifying our whole test suite, just the page object.

Although normally it is used for testing, the Page Object pattern can be used for any purpose related to UI automation—load testing, for example.

Best practices for page objects

The API of the page object must be formulated in terms of the logical structure of the UI and not in terms of the details of the HTML. Basically, the language of the tests and the one in the Page Object's API should be the same. Some best practices to model the API of a page object are as follows:

- Allow read-only accessors to get the information shown by the HTML rendered by the widget. These accessors should not receive any parameters, and they should return a promise containing the value shown, not a WebElement. The accessor should be named after the information we are looking for. For example, `orderView.totalPrice()` would return a promise with a string with the price shown.
- As an alternative to several read-only accessors, use a single one that will return a promise of an object with a field for each information. For example, `orderView.info()` would return a promise of an object with the `totalPrice`, `items`, and other such fields.

- Create read/write accessors for the input fields. The read accessor will return a promise with the current value of the input. The write accessor receives the new value and returns an empty promise that will be fulfilled when the input field's value has been modified. Controls such as checkboxes should model the value as boolean. Multiple choice elements should simply return the values that are selected. Text inputs should be able to receive not only text, but also keys. For example, `quantity()` would return a promise with the value of the quantity field, and `quantity('22')` should send the corresponding text to the input and return an empty promise.
- Sometimes, we need an extra degree of control for inputs. In this case, model the input control with its own small page object, accessible through a read-only accessor. For example, `orderView.addBeverage().quantity()` would return an object with a `value()` read/write accessor and extra methods for interaction, such as `pressEnter()`, or for inspection, such as `isPresent()`, `isVisible()`, or `isMarkedAsError()`.
- Create action methods for buttons, links, and others, named after the action the control represents, that will trigger the action. For example, we can have a `orderView.addBeverage().addToOrder()` method that will issue a click event in the `addToOrder` button of the form. If we need extra control of the interactivity, we can create, as we did earlier, a small page object for the control, instead of a simple action method. Add methods such as `click()`, `pressEnter()`, and `isEnabled()` to it in order to be able to issue events and inspect the control.
- Model each form as a small nested page object. For example, `orderView.addBeverageForm()` should return a page object for that form.
- Create a composite page object if it is necessary, instead of a big giant object.

As you can see, there are two kinds of methods in a page object: query methods, which return the information shown on a page, and action methods, which interact with the page by sending events and filling inputs.

In general, a page object's query method should return the following:

- Another page object, if there is a composition relationship between both or, in other words, if a widget is inside another widget. If necessary, we can consider forms, and even controls, as small page objects. This is the case when we need to interact with them using different gestures, or we are interested in a lot of information, not only in the value of the inputs.
- A promise for a primitive that represents the information shown to the user, or the value of an input.

Action methods are simpler; they simply return a promise that will be fulfilled when the action is complete. If necessary, they can take parameters.

How can we model navigation using page objects? This is a tricky question. One possible design is to make the methods that trigger navigation return the page object for the target of the navigation. For example, if we submit the `placeOrder` form, we should show the UI for the payment; then, we can make `submitPlaceOrder()` return `PaymentPageObject`. However, this is wrong! How do you know what the destination of the `placeOrder` form is? In fact, there should be a test, such as "when we submit the place order form, we go to the payment UI, testing exactly that". Furthermore, almost always, there will be other scenarios, different from the success case; these scenarios will expect you to go to other pages, and not to the payment UI. So, the approach of making the action methods that trigger navigation return another page object is not a good practice, but a common pitfall.

What happens here is that the navigation logic does not belong to the passive view, which is what we are trying to test with the help of the Page Object pattern. Navigation is a concern of the core UI logic, most specifically of the controller or the router components (depending on the design pattern you are using). We can test that kind of logic without WebDriver or any page object, using only vanilla BDD. What we really should test for the passive view is whether it knows how to react correctly to the changes on the current page, and whether it hides and shows the corresponding HTML when it is instructed to do so. We will see how to test this later.



You can end up needing to introduce navigation in the page object if you do integrated tests of the view and the core UI logic. This is a very popular approach but, in the previous chapter, we saw that it is not good. The fact that introducing navigation in a page object is problematic is yet another bad smell of integrated testing.

Another common pitfall is to add assertion methods to a page object. An assertion method would make a check and throw `AssertionError` if the test is not correct. For example, `orderView.addBeverageForm().quantity().assertThatIsMarkedAsError()` would check whether the input control has an `.error` class; if not, it would throw `AssertionError`. A page object should not have assertions. The responsibility of a page object is to access and interact with the page and not check anything. The assertions should be done in the tests themselves.

To sum up, apart from our page objects, we should keep things such as navigation and assertion checking and focus them on doing page interaction and inspection.

Here, we are dealing with an HTML interface, but a good page object will hide the fact that the UI is HTML-based. In principle, if your page object's API is well designed, then it should be possible to change from HTML to another UI technology, such as SVG, native Android, or native iPhone, without changing any of your tests and changing only the implementation of the page object.

Originally, the Page Object pattern was intended to model a whole page. This made sense when the page was the main HTML UI building block. Nowadays, the trend is to create reusable design units, also known as widgets, and build the web pages using them. So, we should not enforce the idea of a single page object per page; rather, we should try to define an individual page object per widget whenever necessary. Furthermore, if a page or widget contains other widgets, you can model it with the composite pattern: a page object composed of other page objects. We can also create a library of page objects not only for these building blocks, but even for smaller controls, such as currency inputs, date pickers, and so on. The goal is to be able to reduce the amount of code we need to write whenever we need to test a new passive view. If we can reuse the existing page objects when we need to test a new passive view, we can save a lot of time and effort. This approach pays only if your application is big enough to leverage these reusable building blocks.

A page object for a rich UI

It is now time to change our UI tests to use the Page Object pattern. The first thing is to create an initial page object that represents the browser. This will encapsulate all the logic about navigation and executing scripts. This way, we can, in the future, replace WebDriver with another tool, if it is necessary. Let's create such an object inside the `test/support/ui.js` file:

```
'use strict';

module.exports = function (port, driver) {
    function uriFor(uiName) {
        return 'http://localhost:' + port + '/test/' + uiName + '.html';
    }

    return {
        uriFor: uriFor,
        goTo: function (uiName) {
            return driver.get(uriFor(uiName));
        },
        executeScript: driver.executeScript.bind(driver),
        executeAsyncScript: driver.executeAsyncScript.bind(driver)
    };
};
```

The module is very simple; it just exposes a constructor for our main page object. The `uriFor` method is almost identical to the one we had before in `test/index.js`, and it will construct the right URI for a given test page. The `goTo` function will instruct WebDriver to open the specified page in the browser, just as we did in the test setup in the previous chapter. The `executeScript` and `executeAsyncScript` functions simply forward the call to the corresponding methods in the WebDriver instance.

This object will be the entry point for our testing utilities, so we need to modify `test/index.js` to expose it to the tests:

```
var express = require('express'),
    port = process.env.PORT || 3000,
    server,
    app = express(),
    browserify = require('browserify'),
    reactify = require('reactify'),
    bundles = {},
    driver,
    webdriver = require('selenium-webdriver'),
    newPageObject = require('./support/ui');

// Skipped for brevity

before('start web driver session', function () {
  driver = new webdriver.Builder().
    withCapabilities(webdriver.Capabilities.chrome()).
    build();

  this.ui = newPageObject(port, driver);
});

after('quit web driver session', function () {
  return driver.quit();
});
// Skipped for brevity
```

We just removed the `beforeEach` block that created the `uriForPage` method, and constructed an instance of the main page object using the WebDriver session.

Building a page object that reads the DOM

Now we can change the `test/order_view_updates_dom.js` feature to use this page object:

```
'use strict';

var chai = require('chai'),
    expect = chai.expect;

chai.use(require('chai-as-promised'));

describe('An order-view updates the DOM', function () {
  var orderView;
  before(function () {
    this.ui.goTo('order');

    orderView = this.ui.newOrderView();

    return orderView.init('.container');
  });

  function willUpdateTheDOM(example) {
    // Skipped for brevity
  }
  // Skipped for brevity
});
```

We just used the `goTo` method of the main page object to open the test page for the order view. Then a new page object was created for the order view, using a factory method called `newOrderView`. The new page object will be stored in the `orderView` variable and will be initialized to be rendered inside the `.container` element. This is exactly the same setup code we had earlier, but the actual code that interacts with WebDriver will have been moved inside the page object. First, we need to add the `newOrderView` method to the main page object in `test/support/ui.js`:

```
var newOrderView = require('../order');

module.exports = function (port, driver) {
  // Skipped for brevity
```

```
    return {
      // Skipped for brevity
      newOrderView: function () {
        return newOrderView(driver);
      }
    };
};
```

Now, we need to create the actual page object for the order view in the `test/support/order.js` file:

```
'use strict';

module.exports = function (driver) {
  var containerSel, self;

  self = {
    init: function (containerSelector) {
      containerSel = containerSelector;
      return driver.executeScript(function () {
        window.view = require('order-view')(arguments[0]);
      }, containerSelector);
    }
  };

  return self;
};
```

Nothing mysterious here; it is the same code we had in the previous chapter but it is neatly encapsulated in the `init` function. Let's add a couple of methods to update the order view with a new view model and to get the total price shown. For this, we just need to move and adapt the code we had in the tests, which is present in the `test/support/` folder:

```
module.exports = function (driver) {
  var containerSel, self;

  self = {
    init: function (containerSelector) {
      containerSel = containerSelector;
      return driver.executeScript(function () {
        window.view = require('order-view')(arguments[0]);
      }, containerSelector);
    },
    update: function (model) {
      // ...
    },
    getTotalPrice: function () {
      // ...
    }
  };

  return self;
};
```

```

update: function (viewModel) {
    return driver.executeAsyncScript(function () {
        // 'done' is callback injected by webdriver as last
        // parameter to notify the async script is done
        var done = arguments[1];
        view.update(arguments[0], done);
    }, viewModel);
},
totalPrice: function () {
    return driver.findElement({
        css: containerSel + ' .order .price'
    }).getText();
}
};

return self;
};

```

Now we can modify the test to use these methods:

```

function willUpdateTheDOM(example) {
    var viewModel = example.viewModel;

    describe('when update is called with ' + example.description,
    function () {
        beforeEach(function () {
            return orderView.update(viewModel);
        });

        it('will update the DOM to show the total price', function () {
            return expect(orderView.totalPrice())
                .to.eventually.be.equal(viewModel.totalPrice);
        });
        // Skipped for brevity
    })
}

```

Now both the action and the test code are more expressive. We can do the same thing with the item list. This time, we can do it a bit differently: when we ask for the items, we can return a slave page object that represents the list of items shown in the order:

```

var newItemView = require('./orderItem');

function newCollection(elements, newView) {
    return {

```

```
size: function () {
  return elements.then(function (arrayOfElements) {
    return arrayOfElements.length;
  });
},
info: function (i) {
  return elements.then(function (arrayOfElements) {
    return newView(arrayOfElements[i]).info();
  });
}
}

module.exports = function (driver) {
  var containerSel, self;

  self = {
    // Skipped for brevity
    items: function () {
      return newCollection(driver.findElements({
        css: containerSel + ' .order .item'
      }), newItemView);
    }
  };

  return self;
};


```

In the `items()` method, we just executed a `findElements` command to get all the elements that represent the items of an order. We passed the result of such a command and a `newItemView` constructor to the collection constructor, `newCollection`.

The page object for the collection has only two methods: `size()` and `info()`. The first will simply return a promise with the length of the array of elements returned by `findElements`. The second will locate a specific element in this array, using the provided index, and will use the `newItemView` to construct a small page object for that element. Then it will use its `info()` method to return the contents of that item.

You can see a composite pattern here: the main page object constructs a page object for the order; we can use the page object for the order to get a page object for the order items collection; and, finally, for each order item in the collection, we can create another page object.



I am very uncomfortable with the name of the pattern: page object. As you can see, it no longer represents a whole page, but a small widget or UI element. That is why, in the code, the variables are not called `*pageObject`, but `*View`. After all, they represent the thing we are testing: the view layer. However, feel free to use any other naming convention that you think is better.

We need to create the page object for each order item and implement its `info()` method. Let's do this inside the `test/support/orderItem.js` file:

```
'use strict';

var promise = require('selenium-webdriver').promise;

module.exports = function (element) {
  return {
    info: function () {
      return promise.all([
        element.findElement({css: '.name'}).getText(),
        element.findElement({css: '.quantity'}).getText(),
        element.findElement({css: '.price'}).getText()
      ]).then(function (fields) {
        return {
          name: fields[0],
          quantity: fields[1],
          unitPrice: fields[2]
        };
      });
    }
  };
};
```

Here, instead of adding a different accessor for each property, we created a single one called `info()`. This method will ask WebDriver for the text content of the name, quantity, and price children elements of this order item element. When this information is available, we will return an object that contains a property for each one of them.

Here, we used the `webdriver.promise.all` utility that waits for all the promises to be fulfilled and returns an array with the values of the promises. We finally transformed this array into a proper object with the `then` method that all promises have.

Now we can implement the test to check the item values:

```
function willUpdateTheDOM(example) {
  var viewModel = example.viewModel;

  describe('when update is called with ' + example.description,
  function () {
    // Skipped for brevity
    describe('will update the DOM to show the items', function () {
      var itemViews;
      beforeEach(function () {
        itemViews = orderView.items();
      });

      it('there is one entry per each item', function () {
        return expect(itemViews.size()).to.eventually
          .be.equal(viewModel.items.length);
      });

      viewModel.items.forEach(function (itemModel, i) {
        it("the DOM for item " + i + " shows the item's information",
        function () {
          return expect(itemViews.info(i))
            .to.eventually.be.deep.equal(itemModel);
        });
      });
    });
  });
}

// Skipped for brevity
}
```

Now we have a single test for the item information instead of three. This is more compact.

Building a page object that interacts with the DOM

We can use the page object to check the form as well. We will use the same strategy: an accessor method in the order page object that will return a page object for the form itself:

```
function willUpdateTheDOM(example) {
  var viewModel = example.viewModel;

  describe('when update is called with ' + example.description,
  function () {
```

```
// Skipped for brevity

describe('will update the DOM to show the add beverage action',
function () {
    var formModel = viewModel.addBeverageForm;
    describe('there is a form', function () {
        beforeEach(function () {
            this.form = orderView.addBeverageForm();
        });

        it('with a ' + formModel.method + ' method', function () {
            return expect(this.form.method())
                .to.eventually.be.equal(formModel.method.toLowerCase());
        });

        it('with action set to ' + formModel.target, function () {
            return expect(this.form.target())
                .to.eventually.match(new RegExp(formModel.target +
'$'));
        });

        it('which is ' + (formModel.shown ? '' : 'not ') + 'visible',
function () {
            return expect(this.form.isShown())
                .to.eventually.be.equal(formModel.shown);
        });

        if (formModel.shown) {
            formModel.messages.forEach(function (msg, i) {
                it('with an error message [' + msg + ']', function () {
                    return expect(this.form.errorMessage(i))
                        .to.eventually.equal(msg);
                });
            });
        }
        // Skipped for brevity
    });
});
});
```

The page object for the form has accessors to return the target URL, the HTTP method to use, whether the form is shown or not, and possible error messages. We can put the implementation of such an object in the `test/support/form.js` file:

```
'use strict';

module.exports = function (driver, element) {
  return {
    method: function () {
      return element.getAttribute('method');
    },
    target: function () {
      return element.getAttribute('action');
    },
    isShown: function () {
      return element.isDisplayed();
    },
    errorMessage: function (i) {
      return element.findElement({
        css: '.error-msg:nth-of-type(' + (i + 1) + ')'
      }).getText();
    }
  };
};
```

The implementation of the methods is again very similar to the one we had in our tests. There is nothing really interesting here. Now, we just need to change the order page object to add an accessor that returns the form page object:

```
var newItemView = require('../orderItem'),
newFormView = require('../form');

// Skipped for brevity

module.exports = function (driver) {
  var containerSel, self;

  self = {
    // Skipped for brevity
    addBeverageForm: function () {
      return newFormView(driver, driver.findElement({
        css: containerSel + ' .order form.add-beverage'
      }));
    }
}
```

```
};

    return self;
};
```

The `addBeverageForm` function will just locate the appropriate form element and build the page object for it.

What about the inputs? Well, we can just continue with our strategy and create a page object for each one:

```
function newInputView(driver, name, element) {
    return {
        name: function () {
            return name;
        },
        type: function () {
            return element.getAttribute('type');
        },
        value: function () {
            return element.getAttribute('value');
        },
        isMarkedAsError: function () {
            return element.getAttribute('class').then(function (classNames)
{
            return classNames.indexOf('error') !== -1;
        });
    },
    isEnabled: function () {
        return driver.executeScript(function () {
            return !arguments[0].disabled;
        }, element)
    }
};
}

module.exports = function (driver, element) {
    return {
        // Skipped for brevity
        fieldWithName: function (name) {
            return newInputView(driver, name, element.findElement({
                css: 'input[name="' + name + '"]'
            }));
        }
    };
};
```

We have added the `fieldWithName` method to the form page object. It will try to locate an input field with the same name inside the form and construct with it an input page object using the `newInputView` factory.

The input page object has methods that allow access to its `name`, `type`, and `value` attributes. The `isMarkedAsError` method encapsulates the logic to know whether the input is highlighted to the user as containing an erroneous value. In our current implementation, it will just check whether the input element has a class named `error` or not. However, if in the future, the HTML design changes, and for example it uses another class name, or any other mechanism, we just need to change this method. This isolates our tests from this implementation detail.

The `isEnabled` method is interesting too. As we saw in the last chapter, the `getAttribute` method of WebDriver cannot be used directly to know whether an input is enabled or not. The HTML attribute that controls this and the corresponding property has the same name, `disabled`, so a call to `getAttribute('disabled')` will only return the value of the attribute. However, the attribute only defines the initial value of the property, and we are interested in its current value and how it changes depending on our test scenarios. That is why we need to use a script to check the JavaScript property directly. All this technical complexity is hidden now in the `isEnabled` method, whereas in the previous chapter it was directly in the tests.

Now we can use this new page object to check the input fields of the form:

```
function willUpdateTheDOM(example) {
    var viewModel = example.viewModel;

    describe('when update is called with ' + example.description,
    function () {
        // Skipped for brevity
        describe('will update the DOM to show the add beverage action',
        function () {
            var formModel = viewModel.addBeverageForm;
            describe('there is a form', function () {
                // Skipped for brevity
                formModel.fields.forEach(function (fieldModel) {
                    var fieldName = fieldModel.name;
                    describe('with a field named ' + fieldName, function () {
                        beforeEach(function () {
                            this.field = this.form.fieldWithName(fieldName);
                        });
                    });
                });
            });
        });
    });
}
```

```

        it('that has type [' + fieldModel.type + ']', function () {
            return expect(this.field.type())
                .to.eventually.be.equal(fieldModel.type);
        });

        it('that has value [' + fieldModel.value + ']', function () {
            return expect(this.field.value())
                .to.eventually.be.equal(fieldModel.value);
        });

        it('that is ' + (fieldModel.error ? '' : 'not ') +
            'highlighted as error', function () {
            return expect(this.field.isMarkedAsError())
                .to.eventually.be.equal (!!fieldModel.error);
        });

        it('that is ' + (formModel.enabled ? 'enabled' :
            'disabled'), function () {
            return expect(this.field.isEnabled())
                .to.eventually.be.equal(formModel.enabled);
        });
    });
}

```

The tests now look much clearer than the one we had in the previous chapter, especially the one regarding whether the field is marked as an error and the one about whether the field is enabled or not. The page object hides all the technical complexity of these tests so that we can focus on making a meaningful and expressive assertion.

Now is the time to update the tests for `test/order_fires_addBeverage.js`:

```

describe('An order-view sends an "add beverage" request to the
controller', function () {
    var addBeverageForm = {
        target: '/orders/items_2',
        method: 'POST',

```

```
enabled: true,
shown: true,
fields: [
  {name: '__method', type: 'hidden', value: 'PUT'},
  {name: 'beverage', type: 'text', value: ''},
  {name: 'quantity', type: 'text', value: ''},
  {name: 'addToOrder', type: 'submit', value: 'Add to order'}
],
messages: []
}, orderView;

before(function () {
  this.ui.goTo('order');

  this.ui.executeScript(function () {
    window.controller = {
      addBeverage: sinon.spy()
    };
  });
}

orderView = this.ui.newOrderView();

orderView.init('.container', 'controller');

orderView.update({
  totalPrice: '0 $',
  items: [],
  addBeverageForm: addBeverageForm
});

this.form = orderView.addBeverageForm();
});

afterEach(function () {
  this.form.fieldWithName('beverage').clear();
  this.form.fieldWithName('quantity').clear();

  this.ui.executeScript(function () {
    window.controller.addBeverage.reset();
  });
});
```

```
function willSendAnAddBeverageRequest(example) {
    // Skipped for brevity
}

// Skipped for brevity
});
```

Although the setup contains almost the same logic as it did earlier, except that it takes advantage of the page object API, there is a subtle point here. We needed to initiate the controller test double; for this, we executed a remote script using the `executeScript` method of our main page object. The new controller test double is stored in the global scope (`window`) under the very imaginative name of `controller`. Now, we needed to pass this test double to our page object to initiate it, but the actual object lives in the browser and not in our test. For this, we added a parameter to the `init` method that contains the name of the global variable that hosts the test double. We can modify the `init` method in `test/support/order.js` as follows:

```
init: function (containerSelector, controllerName) {
    containerSel = containerSelector;
    return driver.executeScript(function () {
        window.view = require('order-view')(arguments[0],
window[arguments[1]]);
    }, containerSelector, controllerName);
}
```

We just passed the name of the variable as a second parameter to the remote script that initializes the order view, and we looked for the test double inside the `window` object.

Now that we have solved the setup, we can have a look at cleaning up the `afterEach` block. We can execute a remote script to reset the test double, but we need to clear the fields of the form too. We can do this if we add a `clear()` method to the input page object:

```
function newInputView(driver, name, element) {
    return {
        // Skipped for brevity
        clear: element.clear.bind(element)
    };
}
```

Since a `WebElement` has a perfectly fine `clear()` method itself, we just delegate to it.



It could be a very good idea to add a `clear()` method to the form that clears all its fields.



In our test, we will need to introduce text into the input, press *Enter*, and click on a button. We can add the corresponding methods to the input page object:

```
var Key = require('selenium-webdriver').Key;

function newInputView(driver, name, element) {
  return {
    // Skipped for brevity
    clear: element.clear.bind(element),
    click: element.click.bind(element),
    typeText: element.sendKeys.bind(element),
    pressKey: function (keyName) {
      return element.sendKeys(Key[keyName]);
    }
  };
}
```

There is nothing very new in the preceding code. Just notice that we have separated the WebDriver's `sendKeys` method into `typeText` and `pressKey`. The intention of the former is to simply type some text into an input element. Although the implementation just delegates to the `sendKeys` method and we do not check for it, we intend only to pass a string as a parameter of `typeText` and not any key. The `pressKey` method is used instead to send a generic key event. For example, `pressKey('SHIFT')` is equivalent to `sendKeys(Key.SHIFT)`. This way, we isolate our tests from the WebDriver API and provide methods with more meaningful names.

With the power of these new methods, we can complete the tests:

```
function willSendAnAddBeverageRequest(example) {
  var enteredBeverage = example.input.beverage,
      enteredQuantity = example.input.quantity;
  describe('given that the user has entered ' + example.title,
  function () {
    var expectedRequest = {
      beverage: enteredBeverage,
      quantity: enteredQuantity,
```

```
target: '/orders/items_2',
method: 'PUT'
};

beforeEach(function () {
    this.form.fieldWithName('beverage')
        .typeText(enteredBeverage);

    this.form.fieldWithName('quantity')
        .typeText(enteredQuantity);
});

it('when the user clicks the "add to order" button, ' +
    'an addBeverage request will be sent to the order with "' +
example.title + '"', function () {
    this.form.fieldWithName('addToOrder').click();

    return this.ui.executeScript(function () {
        expect(controller.addBeverage)
            .to.have.been.calledWith(arguments[0]);
    }, expectedRequest);
});

['beverage', 'quantity'].forEach(function (fieldName) {
    it('when the user press ENTER in the "' + fieldName + '" input,
' +
        'an addBeverage request will be sent to the order with "' +
example.title + '"', function () {
        this.form.fieldWithName(fieldName).pressKey('ENTER');

        return this.ui.executeScript(function () {
            expect(controller.addBeverage)
                .to.have.been.calledWith(arguments[0]);
        }, expectedRequest);
    });
});
});
```

We made our test more compact and legible. Not only that, but note that both the form and input page objects, and the collection page object, can be reused for any other form or collections of elements. The form and input interfaces are very generic, so they can represent any form and set of inputs. It will be perfectly fine for another form, such as placing an order or removing a beverage. Similarly, you can reuse the collection page object for different collections of things; just pass a different page object factory function. This will save us a lot of code across tests, since all the repetitive technical details about the WebDriver API are encapsulated in only one place. This is the main advantage of this way of designing page objects.

As a disadvantage, one can argue that these objects small page objects are not much different from WebDriver's `WebElement` objects. This is true, but we still retain the possibility of replacing WebDriver with an other tool, since the API is now neutral. Furthermore, we have achieved our main goal: to encapsulate knowledge about the HTML structure of our UI inside the page objects. Whether you should use this approach or just try to model each form with a page object with a custom and more abstract API is your decision. Just remember the main goals of using a page object and the tradeoffs involved in your specific case.

Testing the navigation

Until now, we have been testing the ability of a view to update the DOM and transform a DOM event to user operations, but what about navigation? Does it make sense to test the navigation in a single-page application? Actually, yes! Sometimes, the user expects to change from one screen to another, or they expect some views to appear and others to disappear. This can be considered as navigation.

The idea is that certain actions, such as form submissions or links, can trigger a screen change. In a traditional UI, this would trigger a change in the URL and a page refresh. The change in the URL is fine, but the page refresh is not. So, if there is a change of screen, we would like to see the new URL in the browser while updating the DOM to show the new view with client-side logic, instead of refreshing the whole page. Furthermore, changes in the URL triggered by the user when they navigate using their browsing history, or the *Back* button of the browser, should trigger a view change too, without a page refresh if possible.



In the implementation of the view, I am using the HTML5 History API (<http://dev.w3.org/html5/spec-LC/history.html>) that allows me to implement this functionality in around six lines of code. No need to use a full-fledged client-side router library here; after all, the navigation flow logic is in the core UI logic layer. Maybe you will need a router there, but definitely not in passive view. An alternative would be to just use the `window.onhashchange` event (<http://dev.w3.org/html5/spec-LC/history.html#event-hashchange>).

We can extend the passive view interface with a `redirectTo(url)` method. This method should be called by the core UI logic to signal a navigation event as opposed to a refresh of the current view with new data represented by the `update(viewModel)` method. This navigation event can happen in response to any user operation, such as `addBeverage`. In general, the view will call a different method whenever a user operation is triggered, and the core UI logic can respond with an update of the current UI or a `redirectTo`, signaling that we should navigate to another screen.

The implementation of `redirectTo` should change the URL of the browser, and notify the core UI logic when this is done. The core UI logic controller can have a `load(url)` method for this purpose, whose responsibility is to generate a view model for the UI represented by the URL (this will often imply an asynchronous web API call using AJAX or some other technology).



In the proposed design, both passive view and the core UI logic are coordinating themselves through an event dance. This is good because it allows us to easily test both layers in separation and because it provides certain decoupling between layers. However, it can make the behavior a bit difficult to understand. Other kinds of designs are possible; for example, the methods of the core UI logic can return a promise with an object representing the next action to be performed by the view. This makes the flow more clear, but gives the view the additional responsibility of decoding the resulting action. Just feel free to play with new designs and see which one is the best for you!

Now we can try to make a test for this behavior; let's create a `test/order_view_reacts_to_navigation.js` file:

```
'use strict';

var chai = require('chai'),
    expect = chai.expect;

chai.use(require('chai-as-promised'));

describe('An order-view reacts to navigation', function () {
  var orderView;
  beforeEach(function () {
    this.ui.goTo('order');

    this.ui.executeScript(function () {
      window.controller = {
        load: sinon.spy(),
        addBeverage: sinon.spy()
      };
    });
  });

  orderView = this.ui.newOrderView();

  return orderView.init('.container', 'controller');
});

['some/other/page', 'place/order/form'].forEach(function (newPage) {
  var newUrl;
  beforeEach(function () {
    newUrl = this.ui.uriFor(newPage);
  });

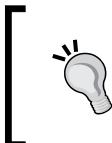
  describe('when redirectTo is called with the URL of ' + newPage,
  function () {
    beforeEach(function () {
      return orderView.redirectTo(newUrl);
    });
  });
});
```

```
it('the browser will not navigate', function () {
    return expect(orderView.isInitialized())
        .to.eventually.be.ok;
});

it('the url will change to the url of ' + newPage, function () {
    return expect(this.ui.currentUrl())
        .to.eventually.be.equal(newUrl);
});

it('a load request will be send to the controller for ' +
newPage, function () {
    return this.ui.executeScript(function () {
        expect(controller.load)
            .to.have.been.calledWith(arguments[0]);
    }, newUrl);
});
// Skipped for brevity
});
});
});
```

We now performed the setup with the `beforeEach` block instead of the `before` block. We did it this way to clean the browser history for each test. Since we are going to play with navigation, the browser history will change its state. However, we need to set up a predictable state in the browser history before each test so that our tests do not interfere with each other, and get the expected result. Unfortunately, WebDriver does not provide a way to clean the history, so we are forced to always reload the order test page. This will not clean the history, but the starting point of the test will always be the same page and the last entries in the history will be always the same.



Alternatively, you can simply use `driver.quit()` after each test and then open a fresh new WebDriver session before each test, with a clean history. However, I do not recommend this because it will make your tests very slow!

Another new thing in the setup is that we added a spy for the `load()` method of the controller test double.

Then we have the action of the test itself; this is performed through the `redirectTo` method of the page object. We still have not added this method to the page object.

Another new method that we need to implement in the page object is the `isInitialized` method. This method will tell us whether the page object represents a valid order view or whether it really cannot find it. It is needed because we need to test whether changing the URL will not trigger a page refresh for the new URL. If this is the case, the code to initialize the order view will not be present in the new page, so this method will return `false`. Let's implement both methods inside `test/support/order.js`:

```
module.exports = function (driver) {
  var containerSel, self;

  self = {
    // Skipped for brevity
    isInitialized: function () {
      return driver.executeScript(function () {
        return window.view && typeof window.view === 'object';
      });
    },
    redirectTo: function (newUrl) {
      return driver.executeScript(function () {
        return window.view.redirectTo(arguments[0]);
      }, newUrl);
    }
  };

  return self;
};
```

The `isInitialized` method just checks that a variable called `view` is defined in the global scope and that it is a non-null object. We can think of more sophisticated checks if necessary, but this is good enough in this example.

The `redirectTo` method will just execute the corresponding method in the view using a remote script.

The `currentUrl` method is not yet implemented. Let's change `test/support/ui.js`:

```
module.exports = function (port, driver) {
  // Skipped for brevity

  return {
    // Skipped for brevity
    currentUrl: driver.getCurrentUrl.bind(driver)
  };
};
```

It simply delegates to the `getCurrentUrl` method of WebDriver.

Now we need to check what happens when the user presses the *Back* button. As we said, we do not really want to navigate; we just want to inform the core UI layer:

```
describe('An order-view reacts to navigation', function () {
    // Skipped for brevity

    ['some/other/page', 'place/order/form'].forEach(function (newPage) {
        // Skipped for brevity
        describe('given there has been a redirection to ' + newPage + ',',
            when the back button is pressed', function () {
                beforeEach(function () {
                    orderView.redirectTo(newUrl);

                    return this.ui.goBack();
                });

                it('the browser will not navigate', function () {
                    return expect(orderView.isInitialized())
                        .to.eventually.be.ok;
                });

                it('a load request will be send to the controller for the
                    previous page', function () {
                    return this.ui.executeScript(function () {
                        expect(controller.load)
                            .to.have.been.calledWith(arguments[0]);
                    }, this.ui.uriFor('order'));
                });
            });
        });
    });
});
```

As we can see, the tests are almost exactly the same. We just made the setup force a redirect to a new page, and then we pressed the *Back* button. In this scenario, the core UI layer should be notified to load the order screen again.

We just need to implement the `goBack` method inside the `test/support/ui.js` file:

```
module.exports = function (port, driver) {
    // Skipped for brevity
    return {
```

```
// Skipped for brevity
goBack: function () {
    return driver.navigate().back();
}
};

};

}
```

We just asked WebDriver to navigate one entry back in the history.

We can continue adding more tests here. For example, we should test whether we hide or remove the DOM for the old view when we change screens.



What happens when the user clicks on a link? In this case, the passive view should notify the core UI logic about a `goToPage(linkTarget)` request, but should abort any possible navigation using `ev.preventDefault()`. It is the same thing with the forms; we check whether the appropriate request has been sent to the core UI logic but, in this case, we do not need to triangulate because links have no input controls.

Summary

You learned that we can prevent maintainability problems in our UI tests with the Page Object pattern: an object that offers a logic and structured view of our UI and hides details about the specific low-level HTML structure. This way, aesthetic changes in the design of the UI can be easily absorbed by our tests, since we do not need to inspect all the tests to make the corresponding changes, only the page object.

Making a page object is not a difficult task; we just need to be a bit careful with its design. A page object should use meaningful names in its methods and stay at the level of abstraction of the UI logical structure, not the raw DOM. You should also try to hide WebDriver from the tests, but do not overdo it. Good examples of this are the `typeText` and `pressKey` methods. Do not couple your page object with the specific framework you are using to implement your passive view, such as React or AngularJS.

Modern web pages are usually composed of several reusable UI building blocks, so it is better to have a reusable page object for each one of these blocks than a big one for a whole page. Also, try to make reusable page objects for common controls, such as the form, the input, and the collection we saw in this chapter. With this approach, you will save code across your tests, and adding a new test will be cheaper.

Finally, keep in mind that the responsibility of the page object is to access and manipulate the browser and the DOM, and not to perform any tests. So, do not add any assertion methods to the page object. This forces us to keep the test double in the tests. Remember that all the assertions are in the tests too, so it is better to be explicit about how the test doubles are constructed, if we need to make an assertion on them.

Page objects should not have any navigation logic. Do not make a method return another page object just because it is supposed to go there if the navigation is successful. Instead, create a specific test suite that defines the navigation.

Now we have a solid foundation to test our UI, and the test code looks better. Unfortunately, we are running the tests only against the Google Chrome browser, and this is not particularly valuable. Remember that passive view tests are expensive, difficult to debug, and slow. If we add the fact that, when we use the correct framework, this layer doesn't have much code, why should we test it at all? The answer lies in cross-browser issues. No browser is 100 percent standards-compliant. They have quirks and weird bugs, so even correct code from the point of view of the standards, that works in a browser, can fail in another one. In the next chapter, we will see how to solve this issue by testing against different browsers.

8

Testing in Several Browsers with Protractor and WebDriver

Until now, we have been doing some quite advanced testing of our UI view layer, but all these tests have been executed against the Google Chrome browser, which is a fairly modern and powerful browser. This is acceptable if we know that our target audience is going to use this browser. Although in some scenarios, such as an internal private tool, this can be the case, in general any application targeted to the public web is going to be executed in a very heterogeneous set of browsers. Since there are not only different levels of adoptions of the HTML5 API, but also subtle bugs through different browsers, we really need to test our view layer in all the browsers that our audience is going to use.

In this chapter, you will see how to run the same test suite against different browsers using two different tools: WebDriver and Protractor, a very popular testing platform for AngularJS. This way, you will be able to take an informed decision about which approach to use in your testing.

Testing in several browsers with WebDriver

There is no need for a special tool to be able to run your test in different browsers. We can do so using WebDriverJS. Let's have a look.

Testing with PhantomJS

We can change our tests to execute them against PhantomJS (<http://phantomjs.org/>). PhantomJS is a headless WebKit that can be accessed and programmed using JavaScript. The point of using a headless browser such as PhantomJS is that it will neither open a window nor render anything on the screen during your tests. Using a headless browser can make your tests run slightly faster, because the browser does not need to open a window process and wait for the HTML to be rendered.



An alternative to using PhantomJS would be to install and configure XVFB (<http://www.x.org/releases/X11R7.6/doc/man/man1/Xvfb.1.xhtml>).



Currently, PhantomJS has built-in support for WebDriver. In the previous versions, there was a need to install a special plugin called GhostDriver for PhantomJS (<https://github.com/detro/ghostdriver>). However, from version 1.8 onwards, it is already built-in in the main distribution of PhantomJS.

The first thing to do is to install the PhantomJS binary in our machine. You can follow the instructions at <http://phantomjs.org/download.html> to do so if you like, but I will do it in another way. I will simply install the NPM `phantomjs` package. This package is a wrapper of PhantomJS that will allow you to access the PhantomJS API from Node.js. As with any other NPM package, we can issue the following command to install it for our project:

```
$ ~/mycafe> npm install --save-dev phantomjs
```

This will install and save the `phantomjs` module as a development dependency of our project. There is no need to install it globally! During the installation, the binaries of the real PhantomJS will be downloaded and built for the specific OS of the machine you are using. This is not only much easier than a manual installation, but also plays better with any CI infrastructure you could have.

Now that we have PhantomJS successfully integrated in our project, it is time to change `test/index.js` in order to instruct WebDriver to use PhantomJS instead of Google Chrome:

```
before('start web driver session', function () {
  driver = new webdriver.Builder().
    WithCapabilities(
```

```

webdriver.Capabilities.phantomjs()
  .set('phantomjs.binary.path', require('phantomjs').path)
) .build();

this.ui = newPageObject(port, driver);
});

```

Note that we only need to change `webdriver.Capabilities.chrome()` to `webdriver.Capabilities.phantomjs()` so that WebDriver can start using PhantomJS. We need to configure the `phantomjs.binary.path` property with the path to the PhantomJS executable. If we do not do so, it will assume that PhantomJS is installed globally in the machine. Fortunately, we can use the `path` property exported by the `phantomjs` module to know where the actual PhantomJS executable has been installed.

Now we can run our tests! Simply issue the `npm test` command as usual. In my case, I implemented the view using ReactJS, and, when I first executed the tests, all were failing. If you run into a similar problem, let me explain what is happening here. The problem is that PhantomJS 1.9's support for **ECMAScript 5 (ES5)** is not good. Some standard libraries of ES5, such as `Function.prototype.bind`, are not supported yet. ReactJS and other modern frameworks rely on the correct implementation of the ES5 standard to work properly.

The fact that the tests fail like this is good, since it forces us to realize that our passive view does not work in older browsers! What can we do? We need to fix our code to work properly in these not-so-modern browsers; for this, we can use a ES5 shim. A shim is a script that implements the missing standard functionality for older browsers so that we can code according to modern standards without worrying about the support issues.



Of course, the use of shims has its limits. Some functionalities cannot be implemented using a small JavaScript library. In these cases, the shim usually requires a native plugin for the browser, and/or the size of the library is too big. This is not the case with regard to the ES5 standard functionality, but it can be a problem for certain HTML5 APIs, such as WebComponents, audio, and so on.

One of the best shims for ES5 is ES5-Shim (<https://github.com/es-shims/es5-shim>). Again, we can directly download scripts from the GitHub project, but it is better to use a prepackaged NPM module. Issuing the `npm install --save-dev es5-shim` command will do the trick.

Now we need to modify our `test/order.html` test page:

```
<!DOCTYPE html>
<html>
<head lang="en">
  <meta charset="UTF-8">
  <title>A test bed page for our Order Passive view</title>
  <link href="/static/css/order.css" type="text/css" rel="stylesheet">
</head>
<body>
<script src="/node_modules/es5-shim/es5-shim.min.js"></script>
  <!-- To be used by injected scripts, normal in browser distribution
  -->
<script src="/node_modules/sinon/pkg/sinon.js"></script>
<script src="/node_modules/chai/chai.js"></script>
<script src="/node_modules/sinon-chai/lib/sinon-chai.js"></script>
<script>expect = chai.expect;</script>
<!-- To be used by common js bundles -->
<script src="/dist/react.js"></script>
<script src="/dist/order-view.js"></script>
<!-- Some markup needed in our test-->
<div class="not-a-container"></div>
<div class="container"></div>
<div class="not-a-container-either"></div>
</body>
</html>
```

We just added a `<script>` tag that will load the `es5-shim.min.js` script from our local `node_modules/` folder.

If we run our tests again, we will see that most of our tests are now passing; however, the ones relating to the method of our form are not. What happens is that PhantomJS is returning the value of the `method` attribute in uppercase. However, Google Chrome returns it in lowercase. We can first change our test in `test/order_view_updates_dom.js`:

```
it('with a ' + formModel.method + ' method', function () {
  return expect(this.form.method())
    .to.eventually.be.equal(formModel.method);
});
```

We replaced `formModel.method.toLowerCase()` with `formModel.method`. This way, the tests pass because we are using uppercase in our test examples, just like PhantomJS. However, now our tests will break if we run them against Google Chrome. To solve this, we need to change our page object for the form in `test/support/form.js`:

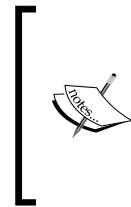
```
module.exports = function (driver, element) {
  return {
    method: function () {
      return element.getAttribute('method')
        .then(function (method) {
          return method.toUpperCase();
        });
    },
    // Skipped for brevity
  };
};
```

We are forcing the page object to return the value of the `method` attribute as uppercase. This is the right thing to do, because the page object should accept and return values that are consistent with the test examples.

Now, our tests are passing in both Chrome and PhantomJS!

Running in several browsers

Now that we know how to persuade WebDriver to use different browsers, we can change our code to make it run the test suite against several browsers with a single command.



If you are using WebDriver 2.44.0 and Firefox 32 or 33, the tests will fail. However, this is not our fault. There was actually a bug in that version of WebDriver. Currently, the fix is planned to be released with version 2.45.0. Take a look at <https://code.google.com/p/selenium/issues/detail?id=8128>.

Since we need to tell Mocha to run the tests several times, we need to change our test files a bit. For example, in `test/order_view_updates_dom.js`:

```
var chai = require('chai'),
    expect = chai.expect;

chai.use(require('chai-as-promised'));

module.exports = function (browserName) {
  describe('[' + browserName + '] An order-view updates the DOM',
    function () {
      // Skipped for brevity
    });
};
```

What we are doing here is transforming the test suite in a Node.js module. This module exports a single test factory function that takes a browser name and creates a test suite for that browser. By default, Mocha will not execute this test suite now; we need to explicitly import this module and call the exported function. We need to make the same change in `test/order_view_fires_addBeverage.js` and `test/order_view_reacts_to_navigation.js`.

Now we need to change the code in `test/index.js`:

```
[

  // Note: Firefox will fail if you use version below 2.45.0
  webdriver.Capabilities.firefox(),
  webdriver.Capabilities.phantomjs()
    .set('phantomjs.binary.path', require('phantomjs').path),
  webdriver.Capabilities.chrome()
].forEach(function (capability) {
  var browserName = capability.get(webdriver.Capability.BROWSER_NAME),
      driver;

  describe('Test suite for [' + browserName + ']', function () {
    before('start web driver session [' + browserName + ']', function () {
      driver = new webdriver.Builder()
        .withCapabilities(capability)
        .build();

      this.ui = newPageObject(port, driver);
    });
});
```

```
require('./order_view_updates_dom')(browserName);
require('./order_view_fires_addBeverage')(browserName);
require('./order_view_reacts_to_navigation')(browserName);

after('quit web driver session [' + browserName + ']',
function () {
    return driver.quit();
});
});
```

We created an array of capabilities and iterated through it. For each capability, we created a different test suite. We did so by importing the test module and invoking the resulting test factory function with the name of the browser. For each suite, a new WebDriver session with its corresponding page object is created.

If we run the tests now, we will see how they are executed for PhantomJS, Firefox, and Google Chrome!

The Selenium Server

For some browsers such as PhantomJS, Google Chrome, or Firefox, we can connect directly to them using the appropriate driver. Unfortunately, WebDriverJS does not support this kind of direct connection to other browsers, such as Safari or Internet Explorer. However, there is a solution for this: using a Selenium Server.

At the time of writing, the latest version of Selenium Server is 2.44.0. Unfortunately, this version has some issues:

- The problem with Firefox that has already been mentioned at <https://code.google.com/p/selenium/issues/detail?id=8128>.
- Some issues with the Selenium Server and the PhantomJS browser. For example, look at *Downgrading to version 2.43.1 fixes the issue* at <https://code.google.com/p/selenium/issues/detail?id=8102>. You can download 2.43.1 from <http://selenium-release.storage.googleapis.com/2.43/selenium-server-standalone-2.43.1.jar>.
- Lack of support for the History API for Safari. Thus, our tests about navigation could fail if you are using this standard API.

The Selenium Server is a process that is able to talk with all the browsers supported by WebDriver. We can connect to it from any kind of process and instruct it to run our commands, using a JSON-based protocol over HTTP (<https://code.google.com/p/selenium/wiki/JsonWireProtocol>). This can be very useful because we can have one or more instances of the Selenium Server running on separate machines with different operating systems and browser versions installed.

 The fact that our test sends commands to the Selenium Server and that the Selenium Server redirects them to the browser, makes our tests a bit slower, because we have an extra latency for this extra connection to the server.

For now, we can keep it simple and just download the Selenium Server to use it with our server. For this, go to <http://www.seleniumhq.org/download/> and download the latest Selenium Server JAR file. Assuming that we have downloaded it to the root of our project, we can issue the following command to start it:

```
$ me@~/mycafe> java -jar selenium-server-standalone-2.44.0.jar -port 4444
```

 For this command to work, you need JAVA installed on your machine.

This command will launch a Selenium Server in the standalone mode. Now, we can modify our test/index.js file to use the server:

```
[  
// Note: Firefox will fail if use version below 2.45.0  
// webdriver.Capabilities.firefox(),  
// webdriver.Capabilities.phantomjs()  
// .set('phantomjs.binary.path', require('phantomjs').path),  
// webdriver.Capabilities.chrome(),  
// webdriver.Capabilities.safari()  
].forEach(function (capability) {  
var browserName = capability.get(webdriver.Capability.BROWSER_NAME),  
    driver;  
  
describe('Test suite for [' + browserName + ']', function () {  
    before('start web driver session [' + browserName + ']', function  
() {  
    driver = new webdriver.Builder()  
        .usingServer('http://localhost:4444/wd/hub')
```

```
.withCapabilities(capability)
.build();

this.ui = newPageObject(port, driver);
});

require('./order_view_updates_dom')(browserName);
require('./order_view_fires_addBeverage')(browserName);
if (browserName !== 'safari') {
    // Version 2.44 or below lacks support for History API for
Safari
    // We will see in next versions...
    require('./order_view_reacts_to_navigation')(browserName);
}
// Skipped for brevity
});
});
```

Note that we have added the capability to test against Safari. We just need to add a line stating that the location of the server is `http://localhost:4444/wd/hub` with the `usingServer()` method.

Launching the Selenium Server manually could be what we want to do if we are using a remote machine for it or if we are using the grid rather than the standalone configuration. In our case, we are not in any of these situations, so it would be nice if WebDriverJS could launch a standalone server on its own and we could avoid this manual step. To do so, we can add the following code to our `test/index.js` file:

```
var express = require('express'),
port = process.env.PORT || 3000,
server,
app = express(),
browserify = require('browserify'),
reactify = require('reactify'),
bundles = {},
webdriver = require('selenium-webdriver'),
newPageObject = require('./support/ui'),
SeleniumServer = require('selenium-webdriver/remote').
SeleniumServer,
seleniumServer;

before('start selenium server', function () {
// If you want to use PhantomJS use version 2.43.1 instead of 2.44
seleniumServer = new SeleniumServer(__dirname +
```

```
'/.../selenium-server-standalone-2.44.0.jar', {
  port: 4444
});

return seleniumServer.start();
});

after('stop selenium server', function () {
  return seleniumServer.stop();
});

// NOTE: If you are using version 2.44 of Selenium Server,
// you cannot test with PhantomJS :( !!!
```

We need to import the `selenium-webdriver/remote` submodule that is built-in in the `selenium-webdriver` module. This module exports a constructor that will allow us to create `SeleniumServer`, start it, and stop it.

To create the instance, we need to pass the path to the JAR file with the implementation of the Selenium Server. We can pass additional parameters, such as the port where the server will listen.

We can now use the Selenium Server object to tell us to which address each new WebDriver session should connect:

```
before('start web driver session [' + browserName + ']', function () {
  driver = new webdriver.Builder()
    .usingServer(seleniumServer.address())
    .withCapabilities(capability)
    .build();

  this.ui = newPageObject(port, driver);
});
```

Welcome Protractor!

Protractor is a wrapper for our test runner that will manage of all the WebDriver details. Actions such as starting, stopping, and configuring the desired capabilities will be taken care of by Protractor. It offers us a more simple and direct way of using WebDriver, one where we need less boilerplate code. On top of that, Protractor wraps the API of WebDriver to make it slightly easier to use.

In the beginning, Protractor was designed to be used to test AngularJS applications using the Jasmine test runner. Currently, we can use Mocha and test applications written with other frameworks.

Let's change our project to use Protractor. We can now remove the `selenium-webdriver` module from our `package.json` file. Do the same thing with its respective folder under `node_modules/`. Remove `chromedriver` and the Selenium Server JAR files. Also remove `test/index.js`. Now we can install Protractor:

```
$ me@~/mycafe> npm install --save-dev protractor
```

The `protractor` module includes all the necessary files and JARs to launch WebDriver. The first thing to do is to tell Protractor to upgrade its WebDriver copy. Then we can start the Selenium Server in standalone mode. Finally, we can tell Protractor to run the tests. For this, we can create some scripts in our `package.json`:

```
"scripts": {
  "upgrade-selenium": "webdriver-manager update",
  "start-selenium": "webdriver-manager start",
  "test": "protractor test/conf.js"
},
```

The `protractor` module includes two executables: `webdriver-manager` and `protractor`. The first allows us to update the local copy of the `chromedriver` and Selenium Server. The second one will run Protractor. Now we can issue the commands to update Selenium and start a standalone server:

```
$ me@~/mycafe> npm run upgrade-selenium
$ me@~/mycafe> npm run start-selenium
```



As we will see shortly, you do not need to start the Selenium Server manually using the `start-selenium` script. If you do not specify the `seleniumAddress` property in the configuration, Protractor will start its own Selenium Server.

Here, we are using `npm run` instead of `npm run-script`; the former is just a shortcut for the latter. With these commands, we should have a running Selenium Server. This is a much simpler approach than the one used earlier. We do not need separate downloads, and everything is handled neatly by Protractor.



Starting the Selenium Server and our development web server should be done better in our build pipeline. We can use GruntJS (<http://gruntjs.com/>), GulpJS (<http://gulpjs.com/>), or Broccoli (<https://github.com/broccolajs/broccoli>) to define a build pipeline that will process the ReactJS files, watch our code and assets, minify, and so on. The thing is that the build pipeline should start and stop the servers whenever required, without the need to do it manually.

Now we need to create a small development web server to serve the test HTML pages and code. Let's create it in `test/support/devServer.js`:

```
'use strict';

var express = require('express'),
    port = process.env.PORT || 3000,
    app = express(),
    browserify = require('browserify'),
    reactify = require('reactify'),
    bundles = {};

function registerBundle(name) {
    return function (err, buf) {
        if (err)
            return console.log(err);
        bundles[name] = buf.toString();
    };
}

var reactFileName = require.resolve('react/dist/react.js');
browserify({
    noParse: [reactFileName]
})
    .require(reactFileName, {expose: 'react'})
    .bundle(registerBundle('react'));

var viewFileName = require.resolve '../../../../../lib/order-view.jsx';

browserify()
    .transform(reactify)
    .require(viewFileName, {expose: 'order-view'})
    .add(viewFileName)
    .exclude('react')
    .bundle(registerBundle('order-view'));

app.use(express.static(__dirname + '/../..'));

app.get('/dist/:bundleName.js', function (req, res) {
    var bundle = bundles[req.param('bundleName')];
    if (!bundle)
        return res.sendStatus(404);
    res.set('Content-Type', 'application/json');
    res.send(bundle);
```

```
});  
  
app.listen(port, function (err) {  
  if (err)  
    return console.log(err);  
  console.log('test server open');  
});
```

This is basically the same code we used earlier, but after removing Mocha and adjusting the paths. Now we can add a new script to our `package.json` file to start this development web server. Our `package.json` file should now look like this:

```
{  
  "name": "mycafe-ui",  
  "version": "0.1.0",  
  "description": "A sample project for testing the UI using  
Protractor",  
  "scripts": {  
    "upgrade-selenium": "webdriver-manager update",  
    "start-selenium": "webdriver-manager start",  
    "start-dev-server": "node ./test/support/devServer.js",  
    "test": "protractor test/conf.js"  
  },  
  "author": "Enrique Amodeo",  
  "license": "MIT",  
  "devDependencies": {  
    "browserify": "^6.1.0",  
    "chai": "^1.9.2",  
    "chai-as-promised": "^4.1.1",  
    "es5-shim": "^4.0.3",  
    "express": "^4.9.8",  
    "mocha": "^1.21.5",  
    "phantomjs": "^1.9.12",  
    "protractor": "^1.4.0",  
    "reactify": "^0.15.2",  
    "sinon": "^1.10.3",  
    "sinon-chai": "^2.6.0"  
  },  
  "dependencies": {  
    "react": "^0.12.0"  
  }  
}
```

Then we can start the server with the following command:

```
$ me@~/mycafe> npm run start-dev-server
```

In order to be able to run our tests, we need to tell Protractor which test files it should run, which test runner to use, the address of the Selenium Server, and which browsers to use. To do so, we need to create a configuration file. According to the test script defined in `package.js`, this file should be `test/conf.js`:

```
exports.config = {
  framework: 'mocha',
  mochaOpts: {
    ui: 'bdd',
    reporter: 'spec',
    timeout: 10000
  },
  seleniumAddress: 'http://localhost:4444/wd/hub',
  specs: [
    './order_*.js'
  ],
  multiCapabilities: [
    // PhantomJS support is broken in Selenium Server 2.44.0
    {
      browserName: 'phantomjs',
      "phantomjs.binary.path": require('phantomjs').path
    },
    {
      browserName: 'chrome'
    },
    {
      browserName: 'safari'
    },
    {
      browserName: 'firefox'
    }
  ],
  onPrepare: function () {
    browser.ignoreSynchronization = true;
  }
};
```

The file is pretty self-explanatory as follows:

- We can control which test runner to use with the `framework` option. Here, it is set to Mocha. Jasmine is the default value. Cucumber is also supported.
- The `mochaOpts` option allows us to pass parameters to Mocha.
- The address of the Selenium Server is specified in the `seleniumAddress` option. If you do not specify this property, Protractor will try to start its own Selenium Server before the tests.
- With the `specs` option, we can define which tests to run using an array of file expressions.
- The `capabilities` option specifies against which browser the Protractor test should run. It consists of an object with at least a `browserName` property that indicates which browser to use. As we will see later, it can have additional properties. This object will be passed as it is to the WebDriver session, so it can contain any special property for each capability. In this case, we are setting the `phantomjs.binary.path` for the `phantomjs` capability.
- If we desire to test against several browsers, we could use the `multicapabilities` option that receives an array of capabilities. This is our case.
- The `onPrepare` option allows us to set some code to be run once before the test is executed, for each capability. This gives us the opportunity to do some customizations and create some utilities for our tests. In this case, we are telling Protractor not to wait for AngularJS to boot. For this, we set the `ignoreSynchronization` property to `true`. This is useful when you are testing a non-AngularJS application, as I am (I'm using ReactJS). If you have used AngularJS to build it, you can safely remove this code.

Now we need to undo the changes we made previously to our test files and add the necessary code to create a page object. For example, `test/order_view_updates_dom.js` will now change in the following way:

```
'use strict';

var chai = require('chai'),
    expect = chai.expect;

chai.use(require('chai-as-promised'));

before('create root page object', function () {
```

```
this.ui = require('./support/ui')(3000, browser.driver);  
});  
  
describe('An order-view updates the DOM', function () {  
    // Skipped for brevity  
});
```

We created a new instance of page object in a `before()` block. Note that Protractor makes a `browser` global variable available; this variable contains the WebDriver session in its `driver` property. We use this property to create our page object. We need to make similar changes to `test/order_view_fires_addBeverage.js` and `test/order_view_reacts_to_navigation.js`.

Finally, we need to change our page objects a bit. We are not installing the `selenium-webdriver` module any more, so we need to remove references to it from our page objects. Fortunately, Protractor gives us a suitable replacement for it.

In `test/support/form.js`, we need to replace the following code:

```
var Key = require('selenium webdriver').Key;
```

The Protractor equivalent is as follows:

```
var Key = protractor.Key;
```

The `test/support/orderItem.js` file needs to be changed as follows:

```
'use strict';  
  
module.exports = function (element) {  
    return {  
        info: function () {  
            return protractor.promise.all([  
                element.findElement({css: '.name'}).getText(),  
                element.findElement({css: '.quantity'}).getText(),  
                element.findElement({css: '.price'}).getText()  
            ]).then(function (fields) {  
                return {  
                    name: fields[0],  
                    quantity: fields[1],  
                    unitPrice: fields[2]  
                }  
            });  
        };  
    };  
};
```

With these changes, our page objects will work with Protractor. Protractor is just a wrapper around WebDriver, so the code we have will work as before.

Now, if we run `npm test`, Protractor will be launched and will execute your tests.

Running the tests in parallel

We can speed up our test suite execution if we run our test suites in parallel. The idea is to run each test file in a separate session of WebDriver, so a different browser window will be opened to run each test file. This can offset the drawback of the additional latency involved in using a Selenium Server.



We can do this easily because our view is stateless! If we were doing traditional end-to-end testing to check our UI, probably we would need a very complex setup to create a server side, or a database with its own set of test data. If not, the setup of each test will mix with each other when they are run in parallel.

Obviously, this is useful when we are using a big server, or a grid of Selenium Server, that can withstand the load of opening and running tests on dozens of browser instances. If you really want to be very sure of your code, you probably would want to run against different browser versions and different operating systems too. So, a Selenium Server grid configuration is necessary in this case.

To configure a grid, you need to start a hub server in one of your boxes:

```
$ me@~/mycafe> java -jar selenium-server-standalone-2.44.0.jar -role hub
```

Then, for each box of your grid, you need to start a Selenium Server instance and register it against the grid:

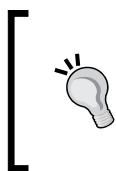
```
$ me@~/mycafe> java -jar selenium-server-standalone-2.44.0.jar -role node
-hub http://mytestserver.com:4444/grid/register
```

Now we can configure the Protractor `test/conf.js` file to run the tests in parallel:

```
seleniumAddress: 'http://mytestserver.com:4444/wd/hub',
multiCapabilities: [
  {
    browserName: 'chrome'
    shardTestFiles: true,
    maxInstances: 4
  },
  ...
]
```

```
{  
  browserName: 'firefox'  
  shardTestFiles: true,  
  maxInstances: 4  
},  
{  
  browserName: 'safari',  
  shardTestFiles: true,  
  maxInstances: 4  
},  
{  
  browserName: 'ie',  
  shardTestFiles: true,  
  maxInstances: 4  
}  
]
```

This configuration will be able to connect to the hub of the grid and run each browser in parallel. For each browser, up to four test files can be run in parallel, as specified in the `maxInstances` property. To make sure that each test file is run in parallel, we must set `shardTestFiles` to `true`.



You can try to launch the tests in parallel without a grid, just with the standalone Selenium Server. This will work, at least for the browsers you have installed on your development machine. I suggest that you set the `maxInstances` option to 2 in order to not overwhelm your development machine.

Other useful configuration options

There are some extra configuration options that can be useful to you; they are as follows:

- If, instead of building your own grid, you are using the Grid in the cloud provided by Sauce Labs (<https://saucelabs.com/selenium/selenium-grid>), you can use the following configuration properties:
 - The `sauceUser` option is used to configure your user
 - The `sauceKey` option is used to configure the key for your account
 - If you are not using the default URL of the Sauce Labs Selenium grid, maybe because you are tunneling through a proxy, you can change the URL with the `sauceSeleniumAddress` option.

- If you do not wish to use a Selenium Server, but rather the drivers for Firefox and Google Chrome, just set the `directConnect` property to `true`. Additionally, you may need to configure the following ones:
 - The `firefoxPath` option is used if your Firefox installation is not in the default path
 - The `chromeDriver` option can be used to tell Protractor where the `chromedriver` file is, if it is not listed in the `PATH` environment variable
- If you want Protractor not to connect to a Selenium Server that is already running but to start its own server, then the `seleniumAddress` property should not be set. You can control this behavior with the following properties:
 - The `port` option will specify in which port to start Selenium Server. The default is `4444`.
 - You can pass other command-line options to the Selenium Server using the `seleniumArgs` property. Its value must be an array of strings that contain the options.
 - Protractor will try to start its own Selenium Server using the binaries packed inside the `protractor` module. To tell Protractor to use different binaries, you can use the `seleniumServerJar` property.
- In addition to the `WebDriver` options and the `shardTestFiles` and `maxInstances` properties, a capability can have a couple more of properties:
 - The `specs` option can contain an array of additional test files to be run only for this capability. These files will be added to the ones specified at the global level.
 - The `exclude` option will remove the specified file from the list of test files to run in this capability. For example, if we cannot run the navigation tests in Safari because the History API is not supported in the current version of `WebDriver`, we can use the following line of code:

```
{  
  browserName: 'safari',  
  exclude: ['test/order_view_reacts_to_navigation.js']  
}
```
- The `jasmineOpts`, `mochaOpts`, and `cucumberOpts` options contain an object with the options to be passed to the corresponding test runner.

- The `getPageTimeout` option controls the timeout in milliseconds for the page load.
- The `allScriptsTimeout` option controls the timeout in milliseconds to wait for the execution of a script.

Using the Protractor API

Now that we have our tests running using Protractor, we can think of changing our page objects to use the API that Protractor offers us, instead of the WebDriverJS API. Our page objects work perfectly fine, but there are two reasons why you would perhaps like to use the Protractor API:

- Your application uses AngularJS, so you need to use the special capabilities that Protractor offers to test AngularJS applications, such as mock modules or searching elements by AngularJS bindings
- Maybe you find the API from Protractor easier to use than the WebDriver one

To sum up, if you are using AngularJS, you can benefit from using the Protractor API instead of the WebDriver API. However, if you are not using AngularJS, it is not so attractive to do so.

Let's start with our main page object in `test/support/ui.js`:

```
'use strict';

var newOrderView = require('./order');

module.exports = function (port, browser) {
  function uriFor(uiName) {
    return 'http://localhost:' + port + '/test/' + uiName + '.html';
  }

  return {
    uriFor: uriFor,
    goTo: function (uiName) {
      return browser.get(uriFor(uiName));
    },
    newOrderView: function () {
      return newOrderView(browser);
    },
    goBack: function () {
```

```
        return browser.navigate().back();
    },
    executeScript: browser.executeScript.bind(browser),
    executeAsyncScript: browser.executeAsyncScript.bind(browser),
    currentUrl: browser.driver.getCurrentUrl.bind(browser.driver)
};

};
```

Now our page object receives and uses the `browser` variable that contains the Protractor instance. The Protractor instance contains many methods that simply decorate the ones in the WebDriver session instance. The main difference is that Protractor methods will wait for AngularJS to be initialized, finish rendering, and, in the case of the navigation methods, it will evaluate the defined AngularJS mock modules we might have defined. So, mostly we just need to replace the `driver` variable with the `browser` instance in most of the methods, and we will be properly integrated with the AngularJS framework. Here, there is really no difference in this example because I am instructing Protractor to explicitly not wait for AngularJS because I am not using it (see the `test/conf.js` file in the previous sections).

The only method that is not exposed by the protractor instance is `getCurrentUrl()`. That is why we need to access the original WebDriver session using `browser.driver`.

In addition, there are some useful methods that will only work if you are using AngularJS, and you should avoid them if you are not. Here they are:

- The `waitForAngular()` method that will return a promise that will be fulfilled when AngularJS has been properly initialized.
- The `setLocation(url)` method will use AngularJS to perform intra-page navigation.
- The `getLocationAbsoluteUrl()` method will ask AngularJS for the current URL, taking into account the intra-page navigation.
- The `addMockModule`, `clearMockModules`, and `removeMockModules` methods will handle the configuration of AngularJS mock modules. These modules can act as test doubles for AngularJS services, controllers, and so on. If we had used AngularJS, they could have been handy in the setup of our tests, since, probably, our core UI logic layer would be an AngularJS module.

Now we can change `test/support/order.js`:

```
'use strict';

var newItemView = require('../orderItem'),
```

```
newFormView = require('./form');

function newCollection(elements, newView) {
  return {
    size: function () {
      return elements.count();
    },
    info: function (i) {
      return newView(elements.get(i)).info();
    }
  }
}

module.exports = function (browser) {
  var totalPrice,
    addBeverageForm,
    items,
    self;

  function initElements(containerSelector) {
    var container = element(by.css(containerSelector));
    totalPrice = container.element(by.css('.order .price'));
    addBeverageForm = container.element(by.css('.order form.add-beverage'));
    items = container.all(by.css('.order .item'));
  }

  self = {
    init: function (containerSelector, controllerName) {
      initElements(containerSelector);

      return browser.executeScript(function () {
        window.view = require('order-view')(arguments[0],
window[arguments[1]]);
      }, containerSelector, controllerName);
    },
    isInitialized: function () {
      return browser.executeScript(function () {
        return window.view && typeof window.view === 'object';
      });
    },
    redirectTo: function (newUrl) {
      return browser.executeScript(function () {
        return window.view.redirectTo(arguments[0]);
      }, newUrl);
    },
    update: function (viewModel) {
      return browser.executeAsyncScript(function () {
```

```
        view.update(arguments[0], arguments[1]);
    },
    viewModel);
},
totalPrice: function () {
    return totalPrice.getText();
},
items: function () {
    return newCollection(items, newItemView);
},
addBeverageForm: function () {
    return newFormView(browser, addBeverageForm);
}
};

return self;
};
```

Here, we have rewritten our order page object using `browser` and `element(locator)`. We use `browser` to execute the needed scripts. The `element(locator)` function is a global function introduced by Protractor that allows us to find elements using a locator parameter. It is the same thing as `driver.findElement(locator)` but will return `ElementFinder` instead of a `WebElement`.

You can think of `ElementFinder` objects as lazy versions of `WebElement`. It has the same methods as `WebElement`, but it will not ask `WebDriver` to find the element until you do not try to interact with it. So, for example you can use the following lines of code:

```
describe('Some test', function() {
    var aButton = element(by.css('button.interesting'));

    beforeEach(function() {
        browser.get('http://localhost:4000/test.html');
    });
    it('when button is clicked, then is cool', function() {
        aButton.click();

        expect(aButton.getAttribute('class'))
            .to.eventually.contain('cool');
    });
});
```

This will work because the button element is not looked for until we try to send a click event to it. In `WebDriver`, it will never work because the moment we search for an element, a command is sent to the browser to try to locate it. So, we are forced to do it in a `beforeEach()` block or in any other context where we already have a `WebDriver` session with a loaded page. Protractor's approach produces more readable code.

As you can see, we are leveraging this mechanism to look for the `ElementFinder` objects, during initialization time, in the `initElements()` helper function. Then, we use those elements to create new page objects in the relevant methods. We first create `ElementFinder` for the container that is the order view. Then we use the `element` and `all` methods of the `ElementFinder` container to locate the other relevant elements. Let's have a look at the most relevant methods of `ElementFinder`:

- The `element(locator)` method receives a locator and returns another `ElementFinder` container with a descendant of the current element that matches the locator. It is exactly the same as the global `element` function, but the global one will look in the whole document instead of only for descendants of a given element. This function will print a warning if the locator matches several descendants.
- The `$(cssSelector)` function is a shortcut to find the set of descendants that match the CSS selector. For example, `element.element(by.css('.error'))` is the same as `element.$('.error')`. There is a global `$` function.
- The `all(locator)` method receives a locator and returns an `ElementArrayFinder` container that represents a collection of elements that are the descendants of this element and matches the locator.
- The `$$($cssSelector)` function is a shortcut to find the set of descendants that match the CSS selector. For example, `element.all(by.css('.error'))` is the same as `element.$$('.error')`. There is a global `$$` function.
- The `getWebElement()` method returns the underlying `WebElement` object for this `ElementFinder` container. Remember that `ElementFinder` is just a lazy wrapper around `WebElement`.
- The `evaluate(expression)` function receives a string with a JS expression and evaluates it using the AngularJS scope associated with the element. It is handy if we intend to access data in that AngularJS scope. Obviously, it will only work if you are using AngularJS otherwise it returns a promise that will be fulfilled with the result of the expression.
- It wraps all the known methods from `WebElement`, such as `click()`, `getText()`, and `sendKeys()`. In most of the context, an `ElementFinder` container can be used exactly as a regular `WebElement`.

It is interesting to know that `ElementFinder` and `ElementArrayFinder` are themselves promises too, so they have a `then` method, and they can be used directly to wait for the element to be located. This implies opening the WebDriver session, loading the test page, and waiting for AngularJS to load and finish rendering.

On the other hand, `ElementArrayFinder` contains methods that we expect to have in a normal array: `map(fn)`, `reduce(fn)`, `each(fn)`, `first()`, `last()`, `get(index)`, and `count()`. All of them return promises with the corresponding result. We have made use of `count()` and `get()` in the `newCollection` page object factory to simplify its implementation.

The locators are almost the same as in WebDriver. There is a global variable called `by` that holds a factory with a different kind of locator. The usual WebDriver locators, such as `by.css()`, `by.id()`, and `by.tagName()`, are present. Additionally, there are specific AngularJS locators, which are follows:

- The `by.model` variable will match any element that has a two-way binding to a given AngularJS model. This is used mostly for input controls, using the `ng-model` attribute.
- The `by.binding` and `by.exactBinding` variables will match any element that has a one-way binding to a given AngularJS model. This is used mostly to update the HTML with an AngularJS model, using the `ng-bind` attribute or the `{...}` shortcut. The `by.binding` version will check whether the `ng-bind` value starts with the value provided. For example, `by.binding('foo')` will match `ng-bind="foobar"`. If you want an exact match, use the `by.exactBinding` version.
- The `by.repeater` variable will find the set of top-level elements inside a container that has a matching `ng-repeat` directive. You can actually use it in four ways:
 - In combination with `element.all`. For example, `element.all(by.repeater('item in order.items'))` will return an `ElementArrayFinder` with one top-level element per order item.
 - Using the `row(index)` sublocator to select a single row in the repeater. For example, `element(by.repeater('item in order.items')).row(1)` will return the top-level element for the second-order item.
 - Using the `column(binding)` sublocator to select only the elements that match the specified binding, instead of the top-level element. For example, `element.all(by.repeater('item in order.items')).column('item.price')` will return an `ElementArrayFinder` with the elements containing the price for each order item, and nothing more.
 - You can use a combination of `row(index)` and `column(binding)`.

Now that we are more familiar with the Protractor API, we can have a look at how to change test/support/orderItem.js:

```
module.exports = function (element) {
  var name = element.$('.name'),
      quantity = element.$('.quantity'),
      price = element.$('.price');

  return {
    info: function () {
      return protractor.promise.all([
        name.getText(),
        quantity.getText(),
        price.getText()
      ]).then(function (fields) {
        return {
          name: fields[0],
          quantity: fields[1],
          unitPrice: fields[2]
        };
      });
    }
  };
};
```

Nothing really advanced here; we only used the \$ function to locate the elements once. The test/support/form.js file is more interesting:

```
var Key = protractor.Key;

function newInputView(browser, name, element) {
  return {
    name: function () {
      return name;
    },
    type: function () {
      return element.getAttribute('type');
    },
    value: function () {
      return element.getAttribute('value');
    },
    isMarkedAsError: function () {
```

```
        return element.getAttribute('class').then(function (classNames)
    {
        return classNames.indexOf('error') !== -1;
    });
},
isEnabled: function () {
    return browser.executeScript(function () {
        return !arguments[0].disabled;
    }, element.getWebElement());
},
clear: element.clear.bind(element),
typeText: element.sendKeys.bind(element),
pressKey: function (keyName) {
    return element.sendKeys(Key[keyName]);
},
click: element.click.bind(element)
};
}
}

module.exports = function (browser, element) {
    var errorMessages = element.$$('.error-msg');

    return {
        method: function () {
            return element.getAttribute('method').then(function (method) {
                return method.toUpperCase();
            });
        },
        target: function () {
            return element.getAttribute('action');
        },
        isShown: function () {
            return element.isDisplayed();
        },
        errorMessage: function (i) {
            return errorMessages.get(i).getText();
        },
        fieldWithName: function (name) {
            return newInputView(browser, name,
                element.$('input[name="' + name + '"]'))
        }
    };
}
```

Again, we used the `$` and `$$` function for brevity. This is very handy for the `errorMessage(i)` method, since we can use `$$` to create an `ElementArrayFinder` for the error messages and use its `get(i)` method to easily implement the functionality.

The `isEnabled()` method of the input page object is tricky. It seems to have almost the same code as it did earlier, but this time we need to retrieve the original `WebElement` using `getWebElement()` to be able to pass it as a parameter of the remote script. Obviously, WebDriver does not know how to serialize an `ElementFinder` and, surprisingly, the wrapper provider by Protractor for the `executeScript` method does not unwrap `ElementFinder` for us.

Summary

In this chapter, you learned how to run our test suite in different browsers. For this, we used two different tools: WebDriver and Protractor.

Although we can directly use WebDriver to do so, the setup and code can be a bit more cumbersome.

Protractor offers us a more streamlined approach than using WebDriver out-of-the-box. The setup and configuration are much easier, and the API that we are offered is more powerful.

This is especially important if your UI is written with AngularJS, since Protractor offers a special API to access and control elements in the context of AngularJS. You can find elements by bindings, model, or repeater. You can also set up test doubles for AngularJS modules, such as controller or service. Finally, Protractor will take care of waiting for AngularJS to finish loading and rendering.

This does not mean that we cannot use Protractor with a non-AngularJS application. As you have seen, we can test a ReactJS application perfectly well using Protractor. You just need to tell Protractor, using the configuration, not to wait for AngularJS. Of course, you need to take care not to use the Protractor API designed to work with AngularJS.

Another point is that, even with our aggressive approach of not doing end-to-end testing, testing against browsers is slow. So, it is a good idea to use a Selenium Server grid and parallelize your test files across the grid. This way, we do not need to wait for a test file to finish before starting another one, and there is no need to wait for the tests to be run in a browser before starting with the next browser. In this aspect, Protractor helps us, since it is very easy to configure to do so.

Finally, I want you to notice all the code and configuration we have written just to test the passive view of our UI. It was a lot of effort, and took us a lot of time. Three full chapters of this book! Of course, we can reuse most of our infrastructure and page objects to make tests cheaper by adding new tests. But, are all these efforts worth enough? After all, if you use a modern framework, then most of the cross-browser issues should be solved by the framework already. Furthermore, testing the core UI logic is much easier and faster. Since the core UI logic should have most of the code of our UI, and the passive view has almost no code, what is the value of exhaustively testing the passive view?

So, think twice before testing the passive view; do it only if you are convinced that these kinds of tests will give you a benefit that justifies all the cost of creating and maintaining them. My general advice is to aim for a very exhaustive test suite of the core UI logic and test only the most problematic aspects of the passive view. However, as always, the tradeoffs of your specific project will tell you what is the correct decision.

9

Testing Against External Systems

In this chapter, we will have a brief look at what happens when we want to test code that depends on another system. How should we approach such a task? Should we do it?

This is a difficult question that has no easy answer, so in this chapter I will show you some techniques that can help you in this subject. This way, you can evaluate whether applying any of these techniques is suitable for your specific case and, if so, whether it can really pay off.

In this chapter, we will learn the following topics:

- Tips on how to write test doubles that are consistent with the expected collaboration between two subsystems or problem domains.
- We will see how to test the code that accesses databases. This technique can be used to test the code that accesses the filesystem or other kinds of storage infrastructure.
- We will explore what happens when we try to test the code that accesses external systems that are not in our control.
- We will see the record-and-replay technique; understanding how it can help us test against external web services.

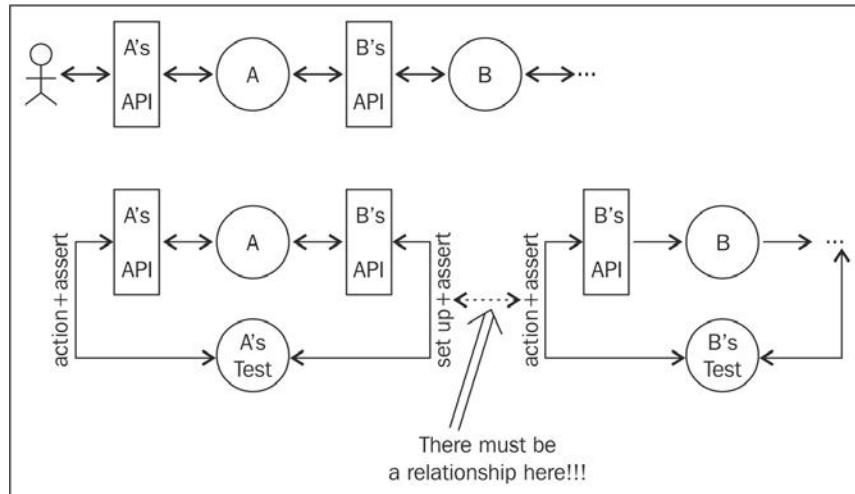
Writing good test doubles

Throughout this book, we have been testing systems that we have been developing to solve our problem domain; hence, they are under our total control. Whenever our system tried to use another one that is not under our control, or simply involves another problem domain, we have used test doubles to replace them in our tests. This is a good practice, since we avoid composing scenarios and features from different systems, and because we can set up such test doubles to perform predictable tests. The trick here is to configure the test doubles to make them behave in a way that simulates the behavior of the other system in a realistic way. This always leads to the same doubt: are our test doubles good enough?

Of course, we will eventually need to write code that implements the interfaces that we have been mocking. This gives us two possibilities, which are explained here:

- We need to write a fair amount of nontrivial code to implement those interfaces. In this case, we can consider that this code is, in fact, a new subsystem, and we can apply the whole BDD approach to it again.
- We actually cannot write much code for it, just a thin layer code that will call an external service, a database, or the platform. This is usually the case with DAOs or service clients. We will see how to deal with this later.

In any case, what should the tests for our new code be like? We need to have some kind of connection between both test suites, because both systems are going to be connected in production. The idea is to do something like what is shown in the following diagram:



Keeping your test doubles honest

In System A, we are using System B, so when we test A, we make test doubles for the interface of B. However, we must ensure that, when we test B, we do so with at least the same calls that A would do to B. The actions performed in the tests of System B and its assertions must have a correlation with the setup we did in the tests of System A.

A simple way to do this is just to use the same test data in both test suites. Let's assume that we have a test in A whose setup says that, when a certain operation on B is called with a parameter, it will return a specific result. In such a case, we should test that, when the same operation is called in B with the same parameters, it will return a result that is consistent with the one we used in the setup.

For example, let's assume that we have the following setup in the test of System A:

```
var b = {
  operation: sinon.spy();
};

b.operation.withArgs(1, 2).returns({
  result: 3
});
```

In this case, the test in the system B should look like this:

```
var result = b.operation(1, 2);

expect(result).to.be.deep.equal({
  args: [1, 2],
  result: 3
});
```

In the test for system B we use the same arguments and the same operation that the one we used in the test double. The assertion checks whether we get an object with a `result` property with the correct value. This check is consistent with what A expects from B and tests whether this part of the contract between A and B is implemented correctly. However, we can see that we are also checking whether the `args` property contains an array with the arguments passed to the operation. This is really not needed at all to check the contract between A and B! This is necessary because B could be used by other systems, not only by A. These other systems can add extra requirements to the API of system B, even if they are not really needed by A.



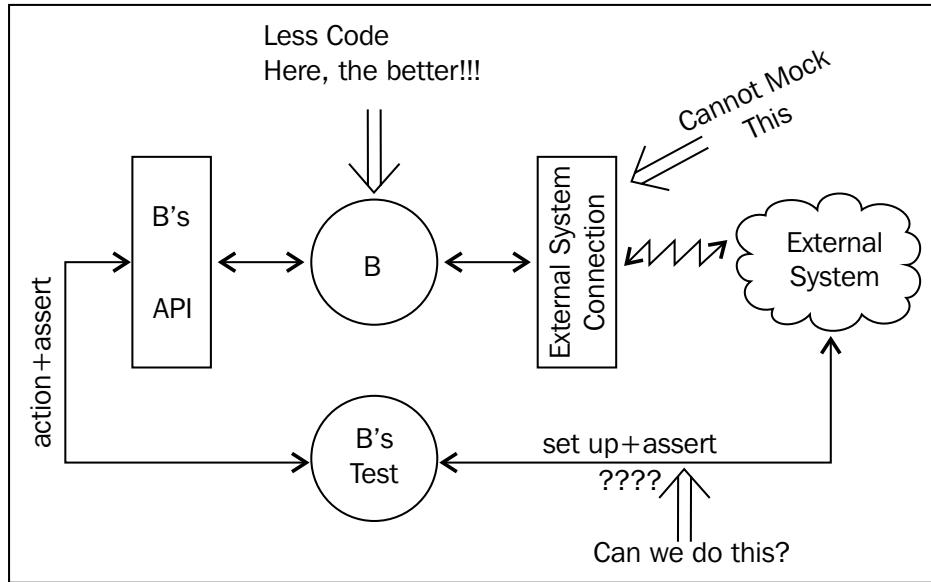
A typical pitfall here is to make the API of B so generic and broad that we end up giving too many responsibilities to B. It is OK that not all the clients use the full API or consume the full data provided by B, only if B solves a single problem domain and its interface is coherent!

What about the setup of B? The setup only depends on the implementation of B and should not have any relationship with tests of A.

Testing against external systems

So far, we have learned a simple technique to maintain the integrity of our test doubles. The problem comes when we need to implement a system that should implement the interfaces represented by these doubles.

Actually, the really problematic case is when we need to test against an external system, because we cannot make a test double for the external system, the database, or the platform. After all, we are trying to test the code layer that talks directly with them. We have reached the boundary not only of our system, but also of our runtime. In these cases, we can only test this code in integration with the external system, so we need an integration test. The following figure illustrates testing against external systems:



Reaching the end of the world

We already saw that the actions and assertions of our tests should be consistent with the setup we made in the tests of the consumer, but how do we set up if we cannot use test doubles again? Well, it depends on which kind of external system we are facing.

Testing against a database

A very common type of external system is a database. This is the case with the Order DAO we had in the earlier chapters. The implementation of such a DAO will only imply making the appropriate calls to the DB client library and configuring the DB connection properly.



The logic of the DAO can sometimes include some simple data adaptation between what is offered by the DB and what is exposed in the DAO interface. However, such logic should be as slim as possible and avoid any transformation at all, whenever we can. The less logic here, the easier it will be to test. Try to move as much logic as possible outside the DAO into the core logic of the application.

All of the techniques we will see in the next two sections can be applied not only to a DB, but also to other kinds of external systems for which we have some kind of a low-level interface—for example, the filesystem.

In our example in *Chapter 3, Writing BDD Features*, we had an Order DAO with two methods: `byId` and `update`. They perform a search by primary key and an update by primary key, respectively. How should we test them? Well, there are mainly two approaches; let's have a look at them.

Accessing the DB directly

When we are writing the DAO, the first thing we need to know is which kind of DB we are going to use. The DAO implementation is going to be totally different if we go for a MySQL than if we use a MongoDB. In any case, whatever DB we are using, it will have some kind of API and connector that we will use to implement our DAO. So, why not use this same DB API to test our DAO?



We are not covering it here, but you will need to install MongoDB and start it before running the tests. As we have seen in the book, this can be done either in a `before()` block or in the project's build pipeline. I am using MongoDB 2.6.x for this example and Version 2.0 of the `mongodb` NPM package. If you do not have MongoDB installed, you can see how to install it at <http://docs.mongodb.org/manual/installation/>.

We can start writing the test with the basic setup:

```
'use strict';

var newOrderDAO = require('../lib/orderDao'),
    MongoClient = require('mongodb').MongoClient,
    ObjectId = require('mongodb').ObjectId,
    expect = require('chai').expect;

describe('An Order DAO', function () {
  var orderDB, config;
  before('connect to DB', function (done) {
    config = {
      port: 27017,
      name: 'order-db'
    };

    MongoClient.connect('mongodb://localhost:' + config.port + '/' + config.name,
      {db: {w: 1}},
      function (err, db) {
        if (err)
          return done(err);
        orderDB = db;
        done();
      }
    );
  });

  after('disconnect from DB', function () {
    orderDB.close();
  });

  beforeEach('clean orders', function (done) {
    orderDB.collection('orders').deleteMany({}, done);
  });

  var orderDAO;
  beforeEach('create dao', function () {
    orderDAO = newOrderDAO(config);
  });

  var theOrder, ordersCollection;
  beforeEach('create test data', function (done) {
```

```
theOrder = {
    _id: new ObjectId(),
    data: [
        {
            beverage: {
                id: "expresso id",
                name: "Expresso",
                price: 1.50
            },
            quantity: 3
        },
        {
            beverage: {
                id: "capuccino id",
                name: "Capuccino",
                price: 2.50
            },
            quantity: 1
        }
    ]
};

ordersCollection = orderDB.collection('orders');
ordersCollection.insertMany([
    {
        _id: new ObjectId(),
        data: []
    },
    theOrder,
    {
        _id: new ObjectId(),
        data: [
            beverage: {
                id: "expresso id",
                name: "Expresso",
                price: 1.50
            },
            quantity: 1
        ]
    }
], done);
});

describe('#byId', function () {
    // TODO
```

```
}) ;

describe('#update', function () {
  // TODO
}) ;
}) ;
```

We are using Version 2.0.x of the `mongodb` package; this is the official MongoDB client for Node.js. The general setup is divided into several stages, which are as follows:

- Opening a connection to the MongoDB. This will be done only once before all the tests, and the resulting connection will be reused throughout the test suite for additional setup and assertions. There is an `after()` block to close the connection when all the tests are done.
- In '`clean orders`', we simply remove all the documents in the `orders` collection before each test. This will ensure that we can make repeatable tests, since we start with a brand new database.
- After cleaning the database, we insert '`create test data`', a set of test orders, into the database. These orders will be used during our tests, and they constitute a baseline against which we can reason about our tests. Note that we store a reference to the order we plan to test against in the `theOrder` variable.
- We finally create an instance of our DAO, pointing to the same database we have set up.

Now that we have the DB in a known state, we can start testing the `byId` method. The success scenario should test whether there is some data in the DB; we only retrieve the document that has the provided ID. We can define an error scenario by saying that, if we do not find the document, we return an error:

```
describe('#byId', function () {
  it('will return the specified order', function (done) {
    var orderId = theOrder._id.toHexString();

    orderDAO.byId(orderId, assertThatSuccessWith(done, function
(order) {
      expect(order).to.be.deep.equal({
        id: orderId,
        data: theOrder.data
      });
    }));
  });
}

it('will return an error if the specified order does not exists',
function (done) {
```

```

var nonExistingId = new ObjectID().toHexString();

orderDAO.byId(nonExistingId, assertThatFailsWith(done,
  function (err) {
    expect(err).to.exist;
    expect(err.toString())
      .to.match(/not found/i)
      .and.to.contain(nonExistingId);
  }));
});
);
);

```

Basically, we implemented the success scenario by simply asking the DAO for `theOrder`, which we know is already inserted into the DB. Then, we checked whether we retrieved the correct result. Note that there is only a small data transformation here: instead of returning a `MongoID` object as the identifier of each order, we used a string as it is expected from the DAO client. The error scenario is similar, but we just tried to retrieve a document that we know does not exist, and we checked whether the returned error contains a useful message.

The only difficulty here is that the API of the DAO is callback-based, using the Node.js convention. This makes our test awkward, because we need to check the result inside the callback. This is a bit tricky and involves some boilerplate code. To simplify the testing, we will use a couple of helper functions:

```

function assertThatSuccessWith(done, assertion) {
  return function (err, result) {
    if (err)
      return done(err);
    try {
      assertion(result);
      done();
    } catch (e) {
      done(e);
    }
  };
}

function assertThatFailsWith(done, assertion) {
  return function (err) {
    try {
      assertion(err);
      done();
    } catch (e) {
      done(e);
    }
  };
}

```

```
        }
    };
}
```

The `assertThatSuccessWith` function takes the done callback from the test and an assertion function, and returns a Node.js callback that we can pass to our DAO's methods. This callback will check whether the DAO method returned an error and, if so, it will inform Mocha of this fact, invoking the done callback with the error. If there is no error, it will execute the assertion function by passing the received result. In this assertion function, we should put all the assertions that we need to check against the result. If everything goes well, the assertion function will not throw, and the done function is called without arguments, telling Mocha that the test passed. If the assertion function throws, it means that the test failed, so we need to call done with the error again. The `assertThatFailsWith` function is very similar, but it just executes the assertion function with the error.

Now we can finish testing the DAO by writing some tests about the `update` method. Again, we have two scenarios: updating an order that already exists and updating a nonexisting order. In the second scenario, we decide that it should not fail, but we insert the nonexisting order as a new one. Let's look at the tests:

```
describe('#update', function () {
  it('will update the specified order', function (done) {
    var orderId = theOrder._id,
      expectedOrder = {
        id: orderId.toHexString(),
        data: theOrder.data.push({
          beverage: {
            id: "mocaccino id",
            name: "Mocaccino",
            price: 4.30
          },
          quantity: 4
        })
      };
    orderDAO.update(expectedOrder, function (err) {
      expect(err).not.to.be.defined;

      ordersCollection.findOne({
        _id: orderId
      }, assertThatSuccessWith(done, function (order) {
        expect(order).to.be.deep.equal({
          _id: orderId,
          data: expectedOrder.data
        });
      }));
    });
  });
});
```

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

it('will create a new order if the specified order does not exists',
  function (done) {
  var orderId = new ObjectId(),
  expectedOrder = {
    id: orderId.toHexString(),
    data: [
      beverage: {
        id: "mocaccino id",
        name: "Mocaccino",
        price: 4.30
      },
      quantity: 4
    ]
  };
}

orderDAO.update(expectedOrder, function (err) {
  expect(err).not.to.exist;

  ordersCollection.findOne({
    _id: orderId
  }, assertThatSuccessWith(done, function (order) {
    expect(order).to.be.deep.equal({
      _id: orderId,
      data: expectedOrder.data
    });
  }));
});
});
```

What changes now is that the assert phase needs to be done directly against the DB. We just wait for the `update` operation to finish and then check whether there are no errors and whether the DB contains the relevant data. This is done with direct access to the DB, using the `monoddb` client library. These tests are a bit verbose, because we need to create the updated version of the order, and we need to nest two callbacks for the assertion. The first callback will wait for the update to finish and check whether there are no errors. Then we need another extra callback to receive the results of the DB, where we check whether the data is correct.

There are some extra best practices that we should observe when testing against a DB; they are as follows:

- Use the same database, versions, configuration, and libraries that are in production.
- It is better to have a test database instance for exclusive use of our test suite.
- If you are using a **relational database**, you must ensure that the schema matches the one expected in your code. For this, there are a couple of best practices, which are as follows:
 - Simply drop the schema and recreate it again. If you are using indexes, do not forget to recreate them.
 - Create the schema with an exclusive name for your test if you can. If it is possible, create a different name whenever your test runs. This implies that you need to inject the schema name to your ORM as a configuration parameter.
 - When testing write operations, always perform the transaction commit phase before asserting. Some DBs will run validation and integrity constraint checks only during a commit. You need to ensure that these checks are passed successfully.
 - Never delete data after a test, but before it. If your test fails, you could just check the contents of the database to debug (assuming you are doing the commit as explained).
 - If you are running your tests in parallel, use separate schema names if using RDBM, or collections names, if using noSQL. As mentioned earlier, you can create a unique name as part of your test setup and pass it to the DAO. This allows your tests to run in parallel without interference. Another option, more expensive, would be to just use different database instances.

Treating the DAO as a collection

If our DAO interface supports a full CRUD API, some of the code we are using in the tests will be suspiciously similar to the one we use in the DAO implementation itself. For example, to test the `byId` method, we need to populate the DB with data during the setup. This setup code is very similar to the one we have to write to implement an `insert/update` method! So this testing approach leads to some duplication of code between tests and production code, which is not good.

A better approach is to treat the DAO like a collection—in this case, a big persistent hash map. If you think about it, you will notice that the logical contract of the DAO is very similar to a hash map. You can even implement the DAO in memory using a JavaScript object—that is, a kind of hash map. So the question is, How would you test a hash map?

The solution here is that you cannot really test each method of the DAO in isolation, but you need to test them in pairs. We can test the query methods doing the setup using the insert, delete, and update methods. Or the other way around, we can test the update method using the query methods in the assertion.

The point is that we are not testing the contract of each method in isolation, because now we cannot have a meaningful contract for each method. In the previous approach, we could define a contract, or behavior, in terms of the inputs of each method and the side-effects in the DB, which we could check directly using the DB client library. Now we cannot check these side-effects, so we need to define the behavior of one method in terms of the others. The contract is defined for the DAO as a whole, and not for each method.

Let's see how we can change the tests now:

```
var orders = [
  {
    id: newId(),
    data: []
  },
  {
    id: newId(),
    data: [
      {
        beverage: {
          id: "expresso id",
          name: "Expresso",
          price: 1.50
        },
        quantity: 3
      },
      {
        beverage: {
          id: "capuccino id",
          name: "Capuccino",
          price: 2.50
        },
        quantity: 1
      }
    ]
  }
]
```

```
        ],
    },
{
  id: newId(),
  data: [
    beverage: {
      id: "expresso id",
      name: "Expresso",
      price: 1.50
    },
    quantity: 1
  ]
}
];

describe('An Order DAO', function () {
  var orderDAO;
  beforeEach(function () {
    orderDAO = newOrderDAO({
      port: 27017,
      name: 'order-db'
    });
  });
  // We will see in a moment what goes here
});
```

Here, we just created some test data and a DAO. Note the use of `newId()` to generate new identifiers for this data. Let's continue:

```
describe('An Order DAO', function () {
  // Skipped for brevity

  describe('Given that there are no orders', function () {
    beforeEach(function (done) {
      orderDAO.removeAll(done);
    });

    function updateAndFindSpec(order) {
      it('when we ask the DAO to retrieve order "' + order.id + '"',
        then an error will be returned', function (done) {
        orderDAO.byId(order.id, assertThatFailsWith(done,
          function (err) {
            expect(err).to.exist;
            expect(err.toString())
              .to.match(/not found/i)
        })
      })
    }
  });
});
```

```

        .and.to.contain(order.id);
    }));
});

it('when we ask the DAO to update order "' + order.id + '", ' +
'then the order "' + order.id + '" can be retrieved',
function (done) {
orderDAO.update(order, function (err) {
expect(err).not.to.exist;

orderDAO.byId(order.id, assertThatSuccessWith(done,
function (result) {
expect(result).to.be.deep.equal(order);
}));
});
});
});
}

orders.forEach(updateAndFindSpec);
});
// Skipped for brevity
});

```

We now added a couple of tests for the `byId` and `update` operations, showing how they behave when they are executed against an empty database. Note how I used the `byId` method to write the assertion in the test for `update`. Let's see what happens when we already have data in the database:

```

describe('An Order DAO', function () {
// Skipped for brevity
describe('Given that we have created three orders', function () {
orders.forEach(function (order) {
beforeEach('insert order "' + order.id + '"', function (done) {
orderDAO.update(order, done);
});
});
}

orders.forEach(function (order) {
it('when the DAO is asked to update "' + order.id + '", ' +
'it will return the new data when retrieved', function (done) {
var newOrderData = {
id: order.id,
data: order.data.push({
beverage: {
id: "latte id",

```

Testing Against External Systems

```
        name: "Latte",
        price: 2.45
    },
    quantity: 3
})
};

orderDAO.update(newOrderData, function (err) {
    expect(err).not.to.exist;

    orderDAO.byId(order.id, assertThatSuccessWith(done,
        function (result) {
            expect(result).to.be.deep.equal(newOrderData);
        }));
    });
});
// Skipped for brevity
});
});
});
```

Let's finish the test suite. For this, we will add the tests for `removeAll` in this last scenario:

```
describe('An Order DAO', function () {
    // Skipped for brevity
    describe('Given that we have created three orders', function () {
        // Skipped for brevity
        describe('when the DAO is asked to remove them all', function () {
            beforeEach(function (done) {
                orderDAO.removeAll(done);
            });

            function assertOrderIsRemoved(order) {
                it('when we ask the DAO to retrieve order "' + order.id + '",',
                    +
                    'then an error will be returned', function (done) {
                        orderDAO.byId(order.id, assertThatFailsWith(done,
                            function (err) {
                                expect(err).to.exist;
                                expect(err.toString())
                                    .to.match(/not found/i)
                                    .and.to.contain(order.id);
                            }));
            });
        });
    });
});
```

```
    }

    orders.forEach(assertOrderIsRemoved);
}) ;
}) ;
}) ;
```

To sum up, we have two basic scenarios, as follows:

- An empty database. Here, we test the `update` methods when called with nonexistent orders, and we also test what happens when we try to get a nonexistent order (since the DB is empty, any order will be nonexistent). The setup is done with the `removeAll` method.
- A database that already contains three orders. We do the setup by inserting the three orders with the `update` method. Here, we test the `removeAll` method, and we also test what happens when we update an existing order.

 Note that we still would need to access the DB directly during setup if we need to drop and recreate the schema. Fortunately, this is not the case in our example.]

In all the tests, we are performing the assertions using the `byId` method, since it is the only way to check the new state of the DB. This forces us to define the behavior of the write methods in terms of `byId`.

Such an approach is powerful, in the sense that we can parameterize these tests and use them for any full CRUD DAO. After all, all of them should have at least this contract. We just need to change the test data set and, maybe, some configuration options.

The main drawback of this approach is that *the tests are not independent of each other*. This is a problem because, if we have a bug in one of the methods, several tests will fail at the same time, and it will be harder to locate the method in which we have the bug. For example, if `byId` has a bug, potentially the whole test suite would fail because we are using it in all the assertions!

Another disadvantage of this is that we cannot implement each method in a separated way; we either implement them all together, or the tests will not pass. This forces us to write most of our tests upfront. This is against the spirit of the test-first development where we aim to write our software incrementally.

Making the assertions for write operations using read operations is not so bad. After all, `byId` is the only way to check what happens to the order list; hence, all the write method contracts are defined in terms of `byId` from a logical point of view.

My main concern is that we are coupling the tests of `removeAll` and `update`. Also note that, in the test for `update`, I need to do the setup using `update` itself! This is definitely not good. We should be able to test `removeAll` independently of the `update` operation, and we should not do the setup using the same method we are testing.

We can do so if we use a direct database call to recreate the database in a known state during setup, as we did in the previous section, and then write all of the assertions only in terms of the read methods of the CRUD. This way, we can make the tests for `update` and `removeAll` independent of each other. Actually, this is the approach I prefer to test a DAO with a full CRUD API. I leave it to you to write such a test suite as an exercise.

Testing against a third-party system

As we have seen, when we are testing against a DB, or a filesystem, we have a certain degree of control, because we can use a low-level API to set up our tests. However, things are not so easy when we test against an external system owned and controlled by a third party.

This is a more common scenario nowadays, since it is common to leverage a web API provided by a third party to enrich and give more value to our application. Furthermore, with the adoption of the micro-services approach, it can happen that these third-party web APIs are, in fact, owned by other teams within your own company!

In general, we cannot directly access the third-party system in order to set it up to a known state. However, almost all of them offer a CRUD API, so we can use the technique that we just saw: using the API itself to perform the setup.

There is only one caveat: the third party must offer us an exclusive and stable test environment. If not, our tests will collide with the ones from other customers of the system. Fortunately, this is not usually a big deal, since most of the service providers offer at least a multitenant environment that we can take advantage of. In the worst case, we should be able to open a test account that is different from the production account to run our tests.

There is an additional concern when testing against a third-party system: unreliable connections. The quality of service of a connection against a third-party service can sometimes be bad. Dropped or slow connections can make our tests slow and unreliable. Tests can fail just because the connection was too slow and not because our code is wrong. This makes us distrust our tests, because they can result in giving us false negatives. At that point, our tests become a liability instead of an investment. Some practitioners claim this as enough reason to not even try to test against third-party systems.

On the other hand, having tests against third-party systems is not only about testing our code, but checking whether we understand the contract of the API. Sometimes, these APIs are not clearly documented, and the communication with the support team can be slow. In this context, these kinds of tests help us explore how to correctly use the API. Another advantage is that, if the contract of the API changes, maybe due to a sudden version change or a bug, our test suite will alert us! These are advantages that, perhaps, we are not willing to lose.

 In a web API, technical details – such as the specific HTTP status code, which mime type it supports, how to perform some operations, or how to discover resources – are part of its contract. Our test suite can protect us against subtle changes in these details.

You should always try to analyze the tradeoffs for your specific situation. The decision about whether to write or not to write tests against third-party systems must be made with care for each specific case.

The record-and-replay testing pattern

There is a technique that can help us with the problems mentioned earlier. This is the record-and-replay testing pattern. In this approach, we replace the third-party system with a fake one. Instead of implementing the fake service ourselves, we can use a library that will record the responses of the server and replay them. This is the workflow for record-and-replay:

1. We run the tests in record mode. Our tests will drive the code that will make calls against the third-party system and will save the responses to a file. In addition to saving the requests and responses, the responses will be also returned to our tests. Sometimes, when we cannot completely set up the third-party system, we cannot predict which responses it will send. In these cases, our tests will fail.
2. We take note of the failures of the tests and check which data the service actually returned. Then, we modify, if necessary, the test suite to expect a result consistent with the real data returned by the third-party system.
3. Run your tests again in the replay mode. If your code is correct, they should pass now. If not, it means that you need to fix the code.
4. Repeat this workflow regularly if you wish, maybe every week, to check that the third-party system API contract has not changed.

There is a big difference with a normal test-first workflow: the setup of the third-party system should not be run in replay mode. In replay mode, our tests will not really act against the third-party system; instead, its calls will be intercepted by the record-and-replay library. There is no sense in running the setup, since all the calls will be intercepted, and their result will be always the same.

There are several modules that can be used here. I will show you a brief example using the `replay` package (<https://github.com/assaf/node-replay>). This package will replace the Node.js `http` package with its own version that will intercept the requests and responses. The record mode will let all of them pass but in addition, it will save the responses for each request in a file. In replay mode, it will use the contents of this file. This implies that `replay` is only useful as long as we use the Node.js HTTP module, either directly or through another library, such as `request`.

Suppose that our product needs to access Twitter, and we want to be able to encapsulate all the logic for it in a small library, based on promises. This way, we can create a test double for it and use it in the tests of our business layer. We can also change the way we access Twitter, via this library – for example, if a new version of the API appears. To test it, we need to create a couple of test user accounts and register our app on Twitter. Our test can be something like this:

```
'use strict';

var newFeed = require('../lib/twitterFeed'),
    chai = require('chai'),
    expect = chai.expect,
    replay = require('replay');

chai.use(require('chai-as-promised'));

describe('A twitter feed', function () {
  var feed;
  beforeEach(function () {
    feed = newFeed({
      // Use your app credentials here!
    });
  });

  function willRetrieveTheLastPublicationsOfAUser(example) {
    var publications = example.expectedPublications.length,
        user = example.user;

    it('will retrieve the last ' + publications + ' publications of @' + user, function () {
```

```

var result = feed.lastTweets(user, numberOfPublications);

return expect(result).to.eventually
  .be.deep.equal(example.expectedPublications);
})
}

[
{
  user: 'mycafeTestX',
  expectedPublications: [
    "insert here contents of tweet 1",
    "insert here contents of tweet 2",
    "insert here contents of tweet 3"
  ]
},
{
  user: 'mycafeTestY',
  expectedPublications: [
    "insert here contents of tweet 1",
    "insert here contents of tweet 2",
    "insert here contents of tweet 3",
    "insert here contents of tweet 4",
    "insert here contents of tweet 5"
  ]
}
].forEach(willRetrieveTheLastPublicationsOfAUser);
});

```

The idea is that our client receives a username and a maximum number of tweets to search, and returns a promise with the last N tweets published by that user. Nothing fancy, we need to change our package.json file to add the test scripts:

```

{
  "name": "replay-record-sample",
  "version": "0.1.0",
  "description": "A sample project of how to test a third party API",
  "main": "index.js",
  "scripts": {
    "record": "REPLAY=record mocha -u bdd -R spec -t 10000 --recursive",
    "test": "mocha -u bdd -R spec -t 100 --recursive"
  },
  "author": "Enrique Amodeo",
}

```

```
"license": "MIT",
"devDependencies": {
  "chai": "^1.10.0",
  "chai-as-promised": "^4.1.1",
  "mocha": "^2.0.1",
  "replay": "^1.11.0"
},
"dependencies": {
  "q": "^1.1.2"
}
}
```

Notice that we installed `replay` as a development dependency. We also created a `record` script that will run our tests in record mode. Now we can issue the `npm run record` command and see our test fail. We can fix this in two different ways, which are explained here:

- Locate the `fixtures/api.twitter.com-443` folder. This folder contains one file per each request/response cycle. In these files, the contents of the requests and responses have been saved during the test execution. We can edit the body of the responses to match the result expected in our tests.
- Simply change the expectations in our test code to match the real data.

Now that we have fixed our test expectations, we can run the test suite in replay mode using `npm test`. Note that the default mode is `replay`, so we do not need to configure the `REPLAY` environment variable. Unfortunately, our tests are still failing! What is happening is that `replay` cannot find a request that matches the ones we are issuing during the tests. The problem is that `replay` uses not only the URL and the query strings to match the requests, but also some HTTP headers. Specifically, it tries to match the content of the following headers:

- Any header starting with `Accept`, `If-`, and `x-`
- The `Authorization` header
- The `Content-Type` header
- The `Host` header

In addition to this, it tries to match the body of the request, if any. So, to match a request, all the headers, body, URL, and query strings must match the recorded request. Only in this case, the response will be replayed.

This is a problem because the Twitter API uses OAuth, and OAuth requires a nonce token, which changes continuously from request to request. This is a typical security feature designed to avoid replay attacks. Since this token is transported in the Authorization header, it makes this header change from request to request, so it will never be matched by the `replay` library.

The solution is to customize which headers to use in the matching process and tell `replay` not to try to match the Authorization header. This can be done by inserting the following lines of code:

```
var newFeed = require('../lib/twitterFeed'),
    chai = require('chai'),
    expect = chai.expect,
    replay = require('replay');

chai.use(require('chai-as-promised'));

replay.headers = [
  /^accept/,
  /^body/,
  /^content-type/,
  /^host/,
  /^if-/,
  /^x-/
];

describe('A twitter feed', function () {
  // Skipped for brevity
});
```

The `headers` property of the `replay` object is an array of regular expressions. Any header that matches one of these regular expressions will be used in the matching algorithm. What I have done here is simply remove the `/^authorization/` regular expression from this array. Now, if you run the tests, they will pass.



You can use this array not only to remove headers, but also to add new headers that you really want to use in the matching algorithm.

This problem about security is not exclusive to the `replay` library, but it is inherent to the record-and-replay approach. This means that this approach is not able to test how our library integrates with the security mechanism of the external service, which is a frequent source of bugs.

Summary

Testing against external systems is expensive! One problem is that setting up the external system in a known state is hard, especially when testing against a third-party system that is not under our control. The other problem is that they can be slow due to connectivity problems.

We can do the setup and assertions using a low-level API or the provided client library directly. This approach generates duplication of code between the test and the production codebase. Instead, I often prefer to use this kind of low-level API only for setup and then write the tests of the DAO, or custom service client, using the read methods of the DAO to write assertions.

If you are testing against a web API, then using the record-and-replay approach is usually a good alternative, especially if the connection is unreliable, slow, or we do not have an exclusive and trustworthy test environment. This adds complexity to our test workflow, since the setup must be generated with a special library that runs the tests in record mode. In addition, some features, such as integration security tokens, OAuth nonce tokens, or CSRF protection tokens, cannot really be tested.

Is all this effort worthy of the result? Consider that the amount of code in the implementation of our DAOs and service clients should be minimal. After all, there should only be a thin wrapper around a service-client library that is already provided, and this is often very easy to code. Furthermore, these tests are not very fast and are subject to glitches, such as slow or unreliable connections, that can make us stop trusting the tests because of the potential false fails. My advice is to think carefully about whether you need to invest time and money in these kinds of tests. If you are going to do it, make sure that you have access to a stable and reliable test environment.

10

Final Thoughts

This is the final chapter of the book; here, you will find a wrap-up of the testing approach presented throughout the book, together with some general advice.
In this chapter:

- We will explore the key difference between TDD and BDD
- We will look at an explanation of what distinguishes a good test from a bad one
- You will learn why integration and end-to-end tests are not good tests
- You will learn the correct granularity for a BDD test
- I will present a summary of which kind of testing approach and which tools you should use to test each part of your system

TDD versus BDD

There are inherently two test-first approaches: **Test Driven Development (TDD)** and **Behavior Driven Development (BDD)**. Some authors consider them to be the same thing, although I don't incline towards this view.

TDD is the earliest of the two approaches and actually only laid the foundations of the test-first cycle. This is not enough, since you can apply the technique at any level of abstraction and in a wide variety of granularity. I have done TDD with a bunch of totally different granularities: testing single classes in isolation, testing private methods, testing a "cluster of objects", and so on. Sometimes, these approaches have been successful, and sometimes not. The key to the success or failure of these tests was that, sometimes, I was testing the right thing, and at other times, I was testing something irrelevant, ending with tons of tests that did not say much about whether the system behaved correctly or not. Does it mean that TDD is wrong (or dead)? No, it just means that we need to constrain its practice to something that makes sense, and here is where BDD fits into the story.

Final Thoughts

Testing whether a low-level component works according to its technical design is not useful. This approach only works in small systems where there is an almost one-to-one correlation between behavior and components. In fact, in small systems such as the ones used in tutorial books, TDD and BDD are very similar. This can prevent readers from appreciating the difference between them.

In a normal system, the relationship between behavior and components is important and really hard to manage. So, when a behavior needs to be changed, all of the test suites give you almost no information about which components need to be changed, what is most important, and how to update the test suite itself to keep it in sync with this change. The information on whether a single feature is implemented or not is scattered across several test suites and not encapsulated in a single test suite. There are two consequences of this, which are as follows:

- Our test suites will slowly get out of sync with the behavior of the system.
- Our tests have low value, even if we have invested a lot of effort in them. They will not tell us which behaviors are broken if one test fails, and they will also not tell us which components need to be updated to accommodate a change in functionality.

For a long time, most of us have been practicing TDD in a way that is totally disconnected from the real value of the software—that is, the functionality of the system (its behavior). We have been thinking that TDD only applies to our work as engineers and that tests are a technical tool and need only check individual low-level technical components (classes or functions).

To solve this situation, BDD proposes that the real thing that we should test is whether the system behaves as expected and not whether a technical component behaves according to some technical design. So, BDD adds a couple of rules on top of TDD:

- Your test suite must check the behaviors that are meaningful for the user of your system, your customer, or any other stakeholder
- Whether a single feature is correctly implemented or not must be checked in a single test suite and not spread over several ones

Both these rules help our test suites to have high value because of the following reasons:

- We are testing the things that our stakeholders really care about
- If a test fails, we can detect which behaviors are broken

- If a test fails, we can detect the cluster of objects that needs to be fixed
- If there is a change in an existing behavior, we can easily locate which tests to change

This ensures an easy correlation between behavior and tests so that both can evolve together. It also ensures that the results of the tests are meaningful to us and the stakeholders.

This does not mean that you should not test individual low-level components, only that you are not forced to do it. If there is a clear value in testing a single low-level component, do it; just do not get carried away and force yourself to test all of them in isolation. Of course, do it in addition to the BDD test suite, not at the expense of it.

The BDD approach sounds easy but, in fact, it is very hard. The problem is that you cannot really test for the behaviors of your system if you do not have a clear understanding of them. Most systems are so big that you need to slice them into smaller parts if you do not want to end up with a big monolith. In practice, we should apply the following additional best practices:

- Locate all the stakeholders of your system and engage with them in a collaborative way to find the features of the system.
- Split your system into different subsystems, one per each problem domain. Each problem domain solves a different problem and has its own language and stakeholders. We have seen several examples of this with the myCafé example; UI, web API, order business layer, payments, and so on are the different problem domains.
- Learn the language of each problem domain and try to define the behaviors in terms of this language. If possible, try to use Cucumber (or similar) to validate your assumptions with the stakeholders and, at the same time, be able to build your test suite on top of it.
- Try to decouple the subsystems as much as possible.

The key to BDD is to be able to interact frequently with the stakeholders, and this can only be done in a truly agile environment. If we cannot do this, then the BDD approach will not work. It is not realistic to ask the engineers to write BDD specs if they cannot interact with the stakeholders to discover and clarify them. In fact, we can stop considering BDD as a testing approach and start thinking of it as an efficient way to capture requisites as executable test suites that both sides, engineers and business people, can understand. Hence, BDD becomes a key practice to bridge the gap between these two collectives and make them work as a team.

A roadmap to BDD

What is the right granularity for a BDD test? This is an important question that I will try to answer in this section.

BDD versus integration testing

There are several misconceptions about BDD, but the most popular one is that BDD implies an integration test of most of the layers of the system. This is actually not true, and it results from a naïve approach to BDD. It is not uncommon to see BDD suites that drive the UI and perform the setup against the DB and the assertion phase using both the UI and the DB. This is not a really good approach, because it results in brittle and slow tests and BDD specifications that try to address all the problem domains at the same time.

A good BDD test suite has the following properties:

- It is repeatable, always giving the same result if there were no changes
- It is not expensive to build
- It is fast, so it gives you feedback quickly
- It gives information that helps you debug an error
- It is easy to change as the requirements change
- It is meaningful for both the stakeholders and the engineers

These properties are the source of the real value of a test suite, and an integration test suite, most of the times, has none of the properties mentioned earlier. As we saw in the previous chapters, these are the reasons why an integration test suite does not have these properties:

- Such a BDD test suite needs to interact with the UI and, usually, with external systems too. Testing the UI is slow, and the tests can be broken if certain changes to the UI are done (even if we do not change business rules). To make it worse, the UI is a part that changes often in any system. Testing against external systems is problematic too. On the one hand, the setup can be difficult to perform. On the other hand, these tests can be nonrepeatable because we do not have total control of the external system.

- What are we really testing with an integration test? The business rules? How the UI behaves? How we communicate with external systems? How we publish our business rules as a web API? The problem is that an integration test suite goes across several problem domains, so we cannot focus on any of them.
- Related to the previous point, in which domain language are we writing our BDD? If we decide to use a single language—for example, the business rules language—we lose a lot of scenarios at the UI level and at the web API level, because we cannot express them in the business rules language. If we decide to mix the languages, then we end up with very unreadable and hard-to-maintain BDD test suites.
- Each distinct problem domain has its own features and scenarios. If we want to test them together and have a high coverage, we need to test all of the combination features and scenarios of all the problem domains. This is clearly not a good idea; it is too expensive.
- If a test fails, which code do I need to change in order to fix the test? An integration test suite gives us no information about the source of the error. It can be any component in any subsystem or layer—or even not in our code, but rather in an external system.

It is much better to try to perform unit tests where each unit under test is tested after being isolated from the other ones and where each behavior is tested independently of the others. Unfortunately, the term unit test is strongly associated with the act of testing individual classes in isolation, so most people will think in a non-BDD approach when talking about unit tests. That is why I prefer to use the term **isolation tests**.

BDD is for testing problem domains

The first step in avoiding the pitfall of integration testing is to realize how you can slice your system into several subsystems, each one taking care of a single problem domain. Each one of these subsystems will have a cohesive language and will serve a subset of our stakeholders, so it is easy to define a very precise and exhaustive BDD feature set for it.



Slicing a system into problem domains is hard, and sometimes we can make a mistake while doing it. However, you can easily detect that you made such a mistake looking for the same problems that an integration test has: complex and hard-to-maintain tests, features that are difficult to understand or write, unclear language, failing tests that are hard to diagnose, and so on. If you start having them, maybe you should refactor your system into smaller problem domains.

The other step is to slice each subsystem into abstraction layers. There will always be the domain layer where we implement a set of rules and data models to solve the domain problem. However, there can be one or more technical layers giving infrastructure services and in charge of nonfunctional requirements, such as persistence, caching, networking, and so on.

We need to architecture and design our system in a way that these layers can be tested independently of one another. This is what we actually have done in this book: testing the passive view separated from the UI control logic or the DAO separated from the order-processing logic.

These technical layers should be ideally very thin, and we can decide not to test them if their code is not complex enough. This is often the case when we are implementing them using a good framework or library that will reduce the complexity of the code. In any case, if you decide to test them, you will not have a clear external stakeholder, but you and your team will be your own stakeholders. In this case, when testing a technical layer, we are not strictly doing BDD, but traditional TDD.

However, sometimes there are technical layers, such as the web API or maybe a security system, can be complex. In these cases, we can separate them as another problem domain and try to use BDD with them. In these cases, the stakeholders are often specialized engineers or architects. However, we can even have normal nonengineers as stakeholders. I can imagine that some people from the business units might have an interest in how the security of your system works, especially in how to manage the access lists and permissions.

A very interesting case is the UI. Clearly, normal users, UX experts, and most business people are strongly interested in how our UI looks and behaves. Also note that how to present data to the user and interact with them is a different problem from how to process transactions in a server according to business rules. It is clear that the UI is a domain problem different from normal business rules, and so it calls for a different subsystem.

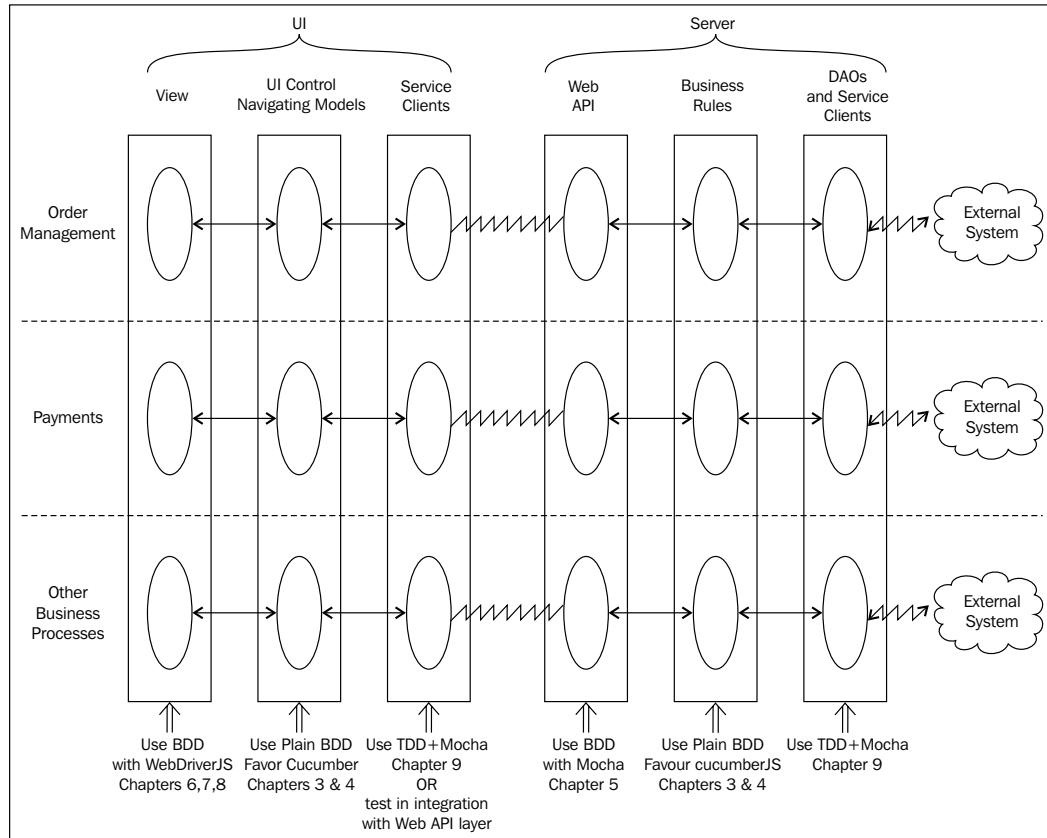
We can also slice the UI into a domain layer, taking care of validation, navigation, and orchestration, and the technical layers of a passive view and the client for the server. We can apply plain BDD to the UI domain layer, but the passive view of the UI should also be addressed by BDD. After all, it transforms gestures to user actions and deals how the UI looks. So, the UX experts, users, and business stakeholders are interested in it. Since there are also a lot of inconsistencies across browsers, the tests for the passive view can have a high value, unless you are using a cross-browser framework that allows you to build this layer very easily.

So, we can slice across problem domains and abstraction layers to obtain a set of subsystems and abstraction layers. For each one of these, we should create a different test suite! For example, in our myCafé example, we can have test suites for the following:

- Order management, payment logic, order UI, payment UI, and so on are subsystems that appear if we slice across the problem domain axis.
- DAOs, service clients, and so on appear if we slice across the abstraction layers. These are details of our technical design, not really a concern to other stakeholders. We can optionally test them, but it is not strictly BDD.
- Order and payment system web APIs, security, and so on are somewhat in the middle. They are technical, but with enough complexity to require a specialist. These specialists become our stakeholders. There can be some interest from nonengineers as well.

Concluding the book

In the following figure, we can see a roadmap to the BDD approach presented in this book:



A BDD roadmap

This figure is just a rough guide; just remember to use your common sense. Developing a test suite implies time and effort; if you do not expect to get enough value in return, just do not do it.

Of course, the architecture of your system may be different from the one I have presented; after all, I have just used one kind of architecture that is popular, but you might have a different one. However, you should always separate your domain layers from the technical ones and slice across functional domains.

Finally, here is a final summary of the whole book:

- Always test the core layers where the domain logic resides. Use plain BDD and favor Cucumber whenever the stakeholders are willing to at least read it. See *Chapter 3, Writing BDD Features* and *Chapter 4, Cucumber.js and Gherkin*.
- It is recommended that you test your web API layer using BDD as explained in *Chapter 5, Testing a REST Web API*. If the interfaces of your business rules layer are consistent enough across business domains, you can create a generic implementation of this layer for all of them.
- It is also recommended that you test your view layer and how it integrates with the DOM. Use BDD and WebDriver as explained in *Chapter 6, Testing a UI Using WebDriverJS*, *Chapter 7, The Page Object Pattern*, and *Chapter 8, Testing in Several Browsers with Protractor and WebDriver*.
- If you are talking with external systems that are out of your control, avoid testing; simply try to do this layer as thin as possible. If you need to test it anyway, see *Chapter 9, Testing Against External Systems*.
- If you are talking with a server that is under your control, for example the service client in the UI, you have more options:
 - Use simple TDD, and replace your network stack with test doubles.
 - Test the service client integrated with the web API layer. In theory, the service client is a proxy of the business rule layer, so it must have the same API. Hence, its implementation is kind of an inverse of the web API layer and must undo its work. So, in a BDD test suite for the client-service layer, you should check whether the service client returns the same result returned by the test double of the business rule layer after traveling through the network.
 - Simply use the techniques in *Chapter 9, Testing Against External Systems*.

Next steps?

What to do next? We have seen many tools and several ways to approach the testing of a whole software application. These techniques are not trivial and usually require some time to master them, but they are not so difficult either! So, my advice is just to try to practice all the techniques presented in this book, in small pet projects at the beginning, where you can make mistakes and fail safely. Then, you can try to start introducing this testing approach in your daily work.

It is important that you get acquainted with these techniques from a practical perspective. Theory is not enough because you always need to answer the following question: Would this test suite give me enough of a return? And you cannot answer this question if you do not have a clear sense of the cost of building the test suite.

Finally, remember that technology changes. Things that nowadays are costly, such as testing the UI, were simply too cumbersome to do in the past, and maybe they will be simple to do in the future. So, stay up-to-date with all the testing tool sets.

We have reached the end of the book! I hope it was useful and that you learned something practical from it.

Summary

We have seen that TDD and BDD are different techniques. BDD is a refinement of TDD; it emphasizes that the important thing is to test behaviors of the system that are relevant to the stakeholders and users.

BDD allows us to write more coherent test suites that can evolve at the same speed as our requirements. A good BDD test suite will give us the ability to track which behaviors are not working correctly in our system and which tests should be changed when we need to change the functionality of the system.

Finally, it is important to distinguish between BDD and integration tests. An integration test checks several problem domains at the same time. This often leads to confused tests that try to test everything and, in the end, are not very effective. Instead, a BDD test suite checks a single problem domain in isolation. This allows the tests to be focused, clear, and easy to write and maintain.

Index

Symbols

200 Ok

API response, testing with 189, 190

-ui option 35

-u option 35

A

addBeverageForm function 287

addMockModule method 323

advanced scenarios

writing 147

advanced setup, Gherkin

about 156, 157

Background section 164

Gherkin-driven example factory 159, 160

implicit setup, versus explicit setup 161-164

afterEach function 63

After hook 175

alert() method 240

all(locator) method 326

allScriptsTimeout option 322

And keyword 133

AngularJS

URL 242

API modeling

best practices 274

API response, with 200 Ok

about 189

empty object, using 190

test, implementing 191, 192

Around hook 175

assertions 42

assert style

URL 45

asynchronous features

callback-based API, testing 89, 90

promise-based API, testing 91

testing 89

B

Background section 164

BDD

about 70, 360

approach 364, 365

properties 360

rules 358

used, for testing problem domains 361, 362

versus integration testing 360, 361

versus TDD 14, 15, 357-359

before function 62

Before hook 174

behavior-driven development. *See* BDD

Broccoli

URL 313

browserify

used, for packing code 245-248

browsers

testing, with WebDriver 303

business layer, HAL resource

using 195-198

But keyword 133

C

callback-based API
testing 89, 90
capabilities option 317
Chai
and Sinon, integrating 73, 74
assertion style, using 41-45
should interface, using 45, 46
chai-as-promised package 97, 98
chains 42
chromeDriver option 321
clear() method 232
clearMockModules method 323
click() method 233
code
cleaning 11, 12
running, in several browsers 307-309
command control flows 239
complex UI interaction
defining 235, 236
configuration options, Protractor
allScriptsTimeout option 322
chromeDriver option 321
cucumberOpts option 321
exclude option 321
firefoxPath option 321
getPageTimeout option 322
jasmineOpts option 321
mochaOpts option 321
port option 321
sauceKey option 320
sauceSeleniumAddress option 320
sauceUser option 320
specs option 321
Continuous Integration (CI) 27, 34
cross-cutting scenarios
extracting 216-219
Cucumber.js
about 130
error reporting 145, 146
features 172
features, tagging 172, 173
hooks 174
non-English Gherkin 176
scenarios, tagging 172, 173

step handler 143-145
using 176, 182
World object pattern 138-142

D

DAO
implementing 120-126
treating, as collection 344-350
dataTable method 156
DB
accessing 337-344
DOM interaction
page object, building with 284-294
DOM, reading
page object, building for 279-283

E

embedded resources
testing 210-216
empty object
using 190
enabled property 255
end-to-end testing 223
error reporting 145, 146
example factory pattern
about 108-112
scenario, finishing 112-115
explicit setup
versus implicit setup 161-164

F

failing test
writing 10-13
fakes, test doubles 23
feature, parameterized scenarios
finishing 170-172
features
adding 219, 220
features, Cucumber.js
tagging 172, 173
fields array 255
findElement(locator) method 235
findElements(locator) method 235
findElements method 231

firefoxPath option 321

flags, Chai

 URL 45

frame(nameOrIndex) method 240

framework option 317

G

getId() method 233

getLocation() method 234

GET order feature

 API response, testing with 200 Ok 189

 exploring 184, 185

 HAL resource, testing for orders 193-195

 server, setting up 186-188

 server, starting 186-188

 server, stopping 186-188

 testing 183, 184

getOuterHtml() method 234

getPageTimeout option 322

getTagName() method 234

getText() method 233

getWebElement() method 326

Gherkin

 about 130

 empty order scenario, steps 153-155

 example tables 147-152

 executing 134-137

 first scenario, writing 132, 133

 project, preparing 130, 131

Gherkin-driven example factory 159, 160

GhostDriver

 URL 304

Given keyword 133

goTo function 278

Grunt

 URL 245

Gulp

 URL 245

H

HAL

 about 179

 URL 180

HAL resource

 business layer, using 195-198

scenario, finishing 198-201

testing, for orders 193-195

handler function 136

hooks

 about 174

 After hook 175

 Around hook 175

 Before hook 174

HTML page

 serving 244

HTML scripts

 serving 244

I

implicit setup

 versus explicit setup 161-164

integration testing

 about 180, 224

 versus BDD 360, 361

Internet application

 setup 242

 testing 241, 242

 UI control logic 267-270

 view reaction, testing to user 260-267

 view, testing for HTML update 249-259

isolation tests 361

isSelected() method 234

K

Knockout

 URL 242

M

method field 255

Mocha

 about 32, 33

 and promises 95-97

 options 35, 36

 test-first cycle 37-41

 URL 36

 using 176, 182

Mocha, options

 -b 35

 --bail option 35

-R 35
--reporter 35
-w 36
--watch option 36
mochaOpts option 317
mocks, test doubles 23
MongoDB
 URL 337
myCafé 77, 78

N

navigation
 testing 294-300
Node
 about 27
 installing 28
 URL 28
Nodebrew
 URL 28
Node Package Manager. *See* **NPM**
Node Version Manager (NVM)
 about 28
 URL 28
non-English Gherkin 176
NPM
 about 27
 installing 28
 project, configuring with 29-32
npm install command 32

O

onPrepare option 317
order actions 202-206, 210
ordering page, myCafé
 features, writing 78
 order, displaying 79-84
 scenarios, coding 86-88
 tips, for writing features 84, 85
orders
 HAL resource, testing for 193-195

P Proudly sourced and uploaded by [StormRG]
package.json
 URL 30

page object
 best practices 274-277
 building, for interacting with DOM 284-294
 building, for reading DOM 279-283
 navigation, testing 294-300
 used, for UI 277, 278
Page Object pattern 273, 274
parameterized scenarios
 about 117-119, 165-169
 feature, finishing 170-172
parameterized tests
 using 56-58
perform method 235
PhantomJS
 URL 304
phases, test
 act 22
 assert 22
 Set up/Arrange 22
port option 321
problem domains
 testing, BDD used 361-363
project
 configuring, with NPM 29-32
promise-based API
 promise 92-95
 testing 91
promises
 about 92-95
 and Mocha 95-97
 test doubles, used with 99, 100
 URL 95
promise, states
 fulfilled 92
 pending 92
 rejected 92
properties, test
 requisites 23, 24
Protractor
 about 312-318
 configuration options 320, 321
 tests, running in parallel 319, 320
 using 322-330
protractor module
 about 313
 protractor 313
 webdriver-manager 313

Q

Q framework

URL 95

R

record and replay pattern 351-355

record and replay tools 223, 224

Red/Green/Refactor

about 47-56

parameterized tests, using 56-58

setup, organizing 60-63

test scenarios, defining 63, 64

redirectTo method 298

regression test suite 8

relational database 344

replay package

URL 352

request module

URL 192

S

sauceKey option 320

Sauce Labs

URL 320

sauceSeleniumAddress option 320

sauceUser option 320

scenario, HAL resource

finishing 198-201

scenario, Cucumber.js

tagging 172, 173

screenshots

taking 240

scripts

injecting 236-238

loading 243

Selenium 2.0 229

seleniumAddress option 317

Selenium Server

about 309-312

URL 310

setup

organizing 58-63

setup, Internet application

about 242

browserify, used for packing code 245-248

HTML page, serving 244

HTML scripts, serving 244

test HTML page 242, 243

WebDriver session, creating 249

shown property 255

Sinon

and Chai, integrating 73, 74

test doubles, used with 65-70

URL 73

using 73

slave resources

cross-cutting scenarios, extracting 216-219

embedded resources, testing 210-215

features, adding 219, 220

order actions 202-210

testing 202

specs option 317, 321

spies, test doubles 23

step handler 143, 145

storage object pattern 104-108

stubs, test doubles 23

submit() method 233

T

target field 255

TDD

about 70, 357

versus BDD 14, 15, 357-359

test

creating, for UI 225

implementing 191, 192

passing 10

running, in parallel 319, 320

test code

example factory pattern 108-112

organizing 101-104

parameterized scenarios 117-119

storage object pattern 104-108

test doubles

about 22, 23

creating 182, 183

fakes 23

mocks 23, 68

spies 23, 68

stubs 23, 68

with promises 99, 100

with Sinon 65-70
writing 334, 335

test-driven development. *See* **TDD**

test-first approach
about 7, 8
test-first cycle 9

test-first cycle
about 9, 37-41
code, cleaning 11, 12
consequences 13
failing test, writing 10
new failing test, writing 12, 13
test, passing 10

test HTML page 242, 243

testing
against database 337
against external systems 336
against third-party system 350, 351
with PhantomJS 304-306

testing, against database
DAO, treating as collection 344-350
DB, accessing directly 337-344

testing, against third-party system
record and replay pattern 351-355

testing architecture, UI 226, 227

testing, with WebDriver
Selenium Server 309-312

test scenarios
defining 63-65

test structure
about 21, 22
test doubles 22, 23

Then keyword 133

Travis
URL 34

triangulation 48

U

UI
page object, used for 277, 278
test, creating for 225

UI control logic 267-270

UI layer
responsibilities 225
UI logic 228
view 228

UI logic component 228

UI testing
end-to-end testing 223, 224
record and replay tools 223
strategy 223, 224
test, creating for UI 225
testing architecture 226-228

unit testing
defining 16-21

V

view
reacting, to user 260-267
testing, for HTML update 249-259

view component 228

Vows
URL 129

W

web API
about 179, 180
Cucumber.js, using 182
Mocha, using 182
responsibilities 180
testing 180, 181
URL 179

webdriver.ActionSequence object
click() 236
doubleClick() 236
dragAndDrop(element, location) 236
keyDown(key) 236
keyUp(key) 236
methods 235
mouseDown() 236
mouseMove(targetLocation, optionalOffset) 236
mouseUp() 236

WebDriver API
URL 231

WebDriverJS
about 229-231
command control flows 238, 239
complex UI interaction, defining 235, 236
screenshots, taking 240

scripts, injecting 236-238
URL 229
used, for controlling frames 240, 241
used, for controlling tabs 240, 241
used, for finding elements 231-234
used, for interacting with elements 231-234
webdriver.Key object 233
WebDriver session
 creating 249
When keyword 133
window.onhashchange event
 URL 295
window(windowName) method 240
World object pattern 138-142

X

XVFB
 URL 304



Thank you for buying
**Learning Behavior-driven Development
with JavaScript**

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.

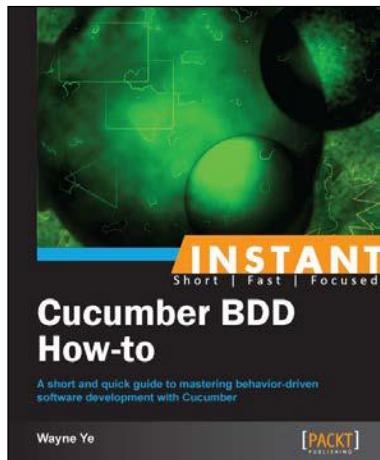
About Packt Open Source

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.

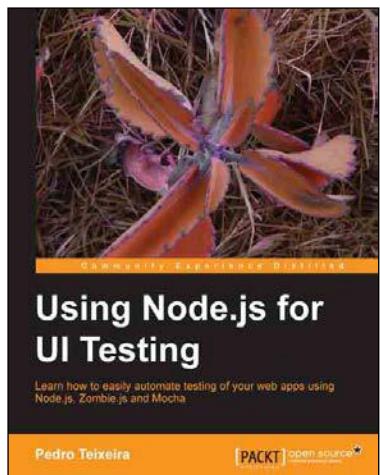


Instant Cucumber BDD How-to

ISBN: 978-1-78216-348-0 Paperback: 70 pages

A short and quick guide to mastering behavior-driven software development with Cucumber

1. Learn something new in an Instant!
A short, fast, focused guide delivering immediate results.
2. A step-by-step process of developing a real project in a BDD-style using Cucumber.
3. Pro tips for writing Cucumber features and steps.
4. Introduces some popular and useful third-party gems used with Cucumber.



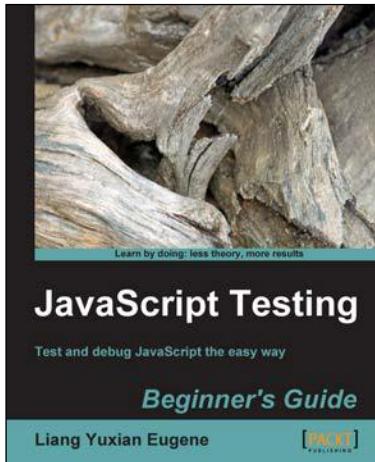
Using Node.js for UI Testing

ISBN: 978-1-78216-052-6 Paperback: 146 pages

Learn how to easily automate testing of your web apps using Node.js, Zombie.js, and Mocha

1. Use automated tests to keep your web app rock solid and bug-free while you code.
2. Use a headless browser to quickly test your web application every time you make a small change to it.
3. Use Mocha to describe and test the capabilities of your web app.

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

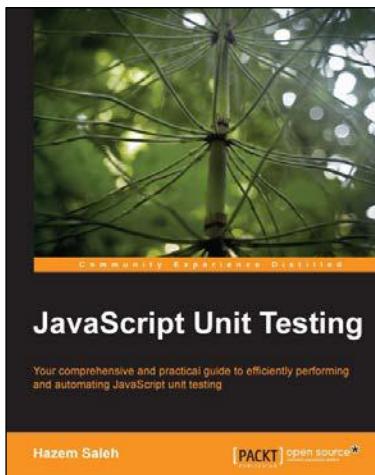


JavaScript Testing Beginner's Guide

ISBN: 978-1-84951-000-4 Paperback: 272 pages

Test and debug JavaScript the easy way

1. Learn different techniques to test JavaScript, no matter how long or short your code might be. Discover the most important and free tools to help make your debugging task less painful. Discover how to test user interfaces that are controlled by JavaScript. Make use of free built-in browser features to quickly find out why your JavaScript code is not working, and most importantly, how to debug it. Automate your testing process using external testing tools.



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.

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