



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

Mastering jQuery

Elevate your development skills by leveraging every available ounce of jQuery

Alex Libby

www.it-ebooks.info

[PACKT] open source*

community experience distilled

Mastering jQuery

Elevate your development skills by leveraging every available ounce of jQuery

Alex Libby



BIRMINGHAM - MUMBAI

Mastering jQuery

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: May 2015

Production reference: 1270515

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

ISBN 978-1-78398-546-3

www.packtpub.com

Credits

Author	Project Coordinator
Alex Libby	Shipra Chawhan
Reviewers	Proofreaders
Islam AlZatary	Stephen Copestake
Ilija Bojchovikj	Safis Editing
Arun P Johny	Joanna McMahon
Lucas Miller	
	Indexer
Commissioning Editor	Tejal Soni
Edward Gordon	
	Production Coordinator
Acquisition Editors	Aparna Bhagat
Vivek Anantharaman	
Owen Roberts	Cover Work
	Aparna Bhagat
Content Development Editor	
Shweta Pant	
Technical Editor	
Tanmayee Patil	
Copy Editors	
Dipti Kapadia	
Jasmine Nadar	
Alpha Singh	

About the Author

Alex Libby has a background in IT support. He has been involved in supporting end users for almost 20 years in a variety of different environments, and he currently works as a technical analyst, supporting a medium-sized SharePoint estate for an international parts distributor who is based in the UK. Although Alex gets to play with different technologies in his day job, his first true love has always been the open source movement, and, in particular, experimenting with CSS/CSS3, jQuery, and HTML5. To date, Alex has already written eight books based on jQuery, HTML5 video, and CSS for Packt Publishing and has reviewed several more. *Mastering jQuery* is Alex's ninth book for Packt Publishing.

I would like to thank my family and friends for their support throughout the process and the reviewers for their valuable comments; this book wouldn't be what it is without them!

About the Reviewers

Islam AlZatary is passionate about new technology in the web industry. He is an entrepreneur and loves to work with smart teams on good ideas. He has a bachelor's degree in computer information system. He has worked for 2 years as a PHP web developer and then he was appointed as a Sr. frontend engineer in 2010.

He deals with jQuery, jQuery UI, HTML/HTML5, CSS/CSS3, the Bootstrap framework, the Mailer template, JavaScript frameworks (RequireJS and AngularJS), and all design approaches. He also likes the mobile-first approach, Magento, e-commerce solutions, and creating his own CSS framework called Lego.

He has reviewed *jQuery UI 1.10: The User Interface Library for jQuery*, Packt Publishing.

He can be found at <http://www.islamzatary.com>. On Twitter, he can be found at <https://twitter.com/islamzatary>. You can also find him on LinkedIn and Facebook.

Ilija Bojchovikj is a talented senior manager of user experience design and development with the proven know-how to combine creative and usability viewpoints resulting in world-class web and mobile applications and systems. He has more than 4 years of experience in partnering with internal and external stakeholders to discover, build, improve, and expand the user experience and create and develop outstanding user interfaces.

He has a proven history of creating cutting edge interface designs and information architectures for websites and mobile applications through a user-centered design process by constructing screen flows, prototypes, and wireframes.

He believes in the power of the Web and those who make it what it is by making and sharing for the sake of betterment; by creating amazing, passionate, and empathetic communities; and by improving the human condition through it all. High five, people of the Web!

You can see what Ilija's up to at <http://Bojchovikj.com> or on Twitter at @il_337.

I'd like to thank the countless people who've given me opportunities and been my inspiration, teachers, sounding boards, my beautiful girlfriend, Monika, and friends. I won't list names because that's just boring and irrelevant for anyone else. Some of you know who you are, others may not. If I've argued passionately with you about the minutiae of nerdy stuff, you're likely to be on the list.

Arun P Johny is a dynamic and competent professional of Web application development with a bachelor's degree in computer application from Kristu Jayanti College, Bangalore. He has been involved in software development for more than 8 years. He is one of the main contributors in Stack Overflow for JavaScript/jQuery questions.

For more than 8 years, he has been associated with Greytip Software Pvt Ltd where he is currently spearheading efforts as the tech lead in charge of developing Greytip Online—the leading online payroll and HR software in India. He focuses on developing a highly scalable and available web application using technologies that are as varied as Java, Spring, and memcache to JavaScript.

Arun has experience in all aspects of software engineering, which include software design, systems architecture, application programming, and testing. He has a vast product development experience in Java, jQuery, Spring, Hibernate, PostgreSQL, and many other enterprise technologies. Arun is skilled in other techniques such as Twitter Bootstrap, AngularJS, ExtJS, FreeMarker, Maven, Spring Security, Git, Activiti, Quartz Scheduler, Eclipse BIRT, and Agile methodology.

Arun loves problem solving and really enjoys brainstorming unique solutions. He can be reached at arunpjohny@gmail.com. You can also get in touch with him at <https://in.linkedin.com/in/arunpjohny>.

I'm greatly thankful to my family members and colleagues for their support and motivation to review this book.

Lucas Miller, Internet Plumber, works as a programmer, designer, and cofounder for the jack-of-all-trades media design group RUST LTD (<http://rustltd.com/>). He likes long walks on the beach, piña coladas, and getting caught in long conversations about the nuances of typeface decisions in popular media. When he isn't developing websites or making games, he can be found teaching, wolf-wrangling, or pursuing whichever new and exciting opportunity presents itself to him that day.

You can find him at <http://lucas.rustltd.com/>.

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	ix
Chapter 1: Installing jQuery	1
Downloading and installing jQuery	1
Using jQuery in a development capacity	2
Adding the jQuery Migrate plugin	3
Using a CDN	4
Using other sources to install jQuery	4
Using NodeJS to install jQuery	5
Installing jQuery using Bower	6
Using the AMD approach to load jQuery	9
Customizing the downloads of jQuery from Git	11
Removing redundant modules	13
Using a GUI as an alternative	14
Adding source map support	14
Adding support for source maps	16
Working with Modernizr as a fallback	16
Best practices for loading jQuery	18
Summary	19
Chapter 2: Customizing jQuery	21
Getting prepared	22
Patching the library on the run	22
Introducing monkey patching	22
Replacing or modifying existing behaviors	23
Creating a basic monkey patch	24
Dissecting our monkey patch	26
Considering the benefits of monkey patching	28

Table of Contents

Updating animation support in jQuery	29
Exploring the requestAnimationFrame API's past	30
Using the requestAnimationFrame method today	30
Creating our demo	30
Adding WebP support to jQuery	34
Getting started	34
Creating our patch	35
Taking things further	39
Considering the pitfalls of monkey patching	41
Distributing or applying patches	42
Summary	44
Chapter 3: Organizing Your Code	45
Introducing design patterns	46
Defining design patterns	46
Dissecting the structure of a design pattern	48
Categorizing patterns	49
The Composite Pattern	49
Advantages and disadvantages of the Composite Pattern	50
The Adapter Pattern	51
Advantages and disadvantages of the Adapter Pattern	51
The Facade Pattern	52
Creating a simple animation	53
Advantages and disadvantages of the Façade Pattern	54
The Observer Pattern	55
Advantages and disadvantages of the Observer Pattern	56
Creating a basic example	57
The Iterator Pattern	58
Advantages and disadvantages of the Iterator Pattern	60
The Lazy Initialization Pattern	61
Advantages and disadvantages of the Lazy Initialization Pattern	61
The Strategy Pattern	62
Building a simple toggle effect	63
Switching between actions	64
Advantages and disadvantages of the Strategy Pattern	64
The Proxy Pattern	65
Advantages and disadvantages of the Proxy Pattern	66
Builder Pattern	67
Advantages and disadvantages of the Builder Pattern	68
Exploring the use of patterns within the jQuery library	69
Summary	72

Table of Contents

Chapter 4: Working with Forms	73
Exploring the need for form validation	74
Creating a basic form	75
Starting with simple HTML5 validation	76
Using HTML5 over jQuery	78
Using jQuery to validate our forms	79
Validating forms using regex statements	81
Creating a regex validation function for e-mails	82
Taking it further for URL validation	83
Building a simple validation plugin	84
Developing a plugin architecture for validation	88
Creating our basic form	89
Creating custom validators	90
Localizing our content	94
Centralizing our error messages	95
Wrapping up development	96
Noting the use of best practices	97
Providing fallback support	98
Creating an advanced contact form using AJAX	99
Developing an advanced file upload form using jQuery	102
Summary	104
Chapter 5: Integrating AJAX	107
Revisiting AJAX	108
Defining AJAX	109
Creating a simple example using AJAX	110
Improving the speed of loading data with static sites	113
Using localStorage to cache AJAX content	115
Using callbacks to handle multiple AJAX requests	117
Enhancing your code with jQuery Deferreds and Promises	118
Working with Deferreds and Promises	120
Modifying our advance contact form	122
Adding file upload capabilities using AJAX	125
Examining the use of Promises and Deferreds in the demo	127
Detailing AJAX best practices	128
Summary	129
Chapter 6: Animating in jQuery	131
Choosing CSS or jQuery	132
Controlling the jQuery animation queue	134
Fixing the problem	135

Table of Contents

Making the transition even smoother	136
Using a pure CSS solution	138
Improving jQuery animations	139
Introducing easing functions	140
Designing custom animations	141
Converting to use with jQuery	143
Implementing some custom animations	145
Animating rollover buttons	145
Exploring the code in more detail	146
Animating an overlay effect	147
Animating in a responsive website	149
Considering animation performance on responsive sites	152
Handling animation requests on a responsive site	154
Animating content for mobile devices	157
Improving the appearance of animations	159
Implementing responsive parallax scrolling	161
Building a parallax scrolling page	161
Considering the implications of parallax scrolling	164
Summary	166
Chapter 7: Advanced Event Handling	167
Introducing event handling	167
Delegating events	168
Revisiting the basics of event delegation	168
Reworking our code	170
Supporting older browsers	170
Exploring a simple demonstration	171
Exploring the implications of using event delegation	172
Controlling delegation	173
Using the stopPropagation() method as an alternative	175
Using the \$.proxy function	177
Creating and decoupling custom event types	180
Creating a custom event	181
Working with the Multiclick event plugin	183
Namespacing events	184
Summary	187
Chapter 8: Using jQuery Effects	189
Revisiting effects	189
Exploring the differences between animation and effects	190
Creating custom effects	190
Exploring the animate() method as the basis for effects	191

Table of Contents

Putting custom effects into action	192
Creating a clickToggle handler	192
Sliding content with a slide-fade Toggle	194
Applying custom easing functions to effects	196
Adding a custom easing to our effect	197
Using Bezier curves in effects	199
Adding Bezier curve support	200
Using pure CSS as an alternative	202
Adding callbacks to our effects	204
Controlling content with jQuery's Promises	205
Creating and managing the effect queue	207
Summary	209
Chapter 9: Using the Web Performance APIs	211
An introduction to the Page Visibility API	211
Supporting the API	212
Implementing the Page Visibility API	212
Breaking down the API	214
Detecting support for the Page Visibility API	214
Providing fallback support	217
Installing visibility.js	217
Building the demo	218
Using the API in a practical context	220
Pausing video or audio	220
Adding support to a CMS	221
Exploring ideas for examples	223
Introducing the requestAnimationFrame API	224
Exploring the concept	225
Viewing the API in action	226
Using the requestAnimationFrame API	226
Retrofitting the changes to jQuery	227
Updating existing code	228
Some examples of using requestAnimationFrame	229
Creating a scrollable effect	229
Animating the Google Maps marker	231
Exploring sources of inspiration	232
Summary	233
Chapter 10: Manipulating Images	235
Manipulating colors in images	235
Adding filters using CSS3	236
Getting ready	237
Creating our base page	237

Table of Contents

Changing the brightness level	240
Adding a sepia filter to our image	240
Exploring other filters	241
Blending images using CSS3	242
Applying filters with CamanJS	244
Introducing CamanJS as a plugin	244
Building a simple demo	244
Getting really creative	246
Creating simple filters manually	248
Grayscaleing an image	249
Adding a sepia tone	251
Blending images	253
Animating images with filters	255
Introducing cssAnimate	255
Creating a signature pad and exporting the image	258
Capturing and manipulating webcam images	260
Finishing up	265
Summary	265
Chapter 11: Authoring Advanced Plugins	267
Detecting signs of a poorly developed plugin	267
Introducing design patterns	269
Creating or using patterns	270
Designing an advanced plugin	271
Rebuilding our plugin using boilerplate	272
Converting animations to use CSS3 automatically	276
Working with CSS-based animations	278
Considering the impact of the change	279
Falling back on jQuery animations	280
Extending our plugin	282
Packaging our plugin using Bower	283
Automating the provision of documentation	285
Returning values from our plugin	286
Exploring best practices and principles	289
Summary	291
Chapter 12: Using jQuery with the Node-WebKit Project	293
Setting the scene	294
Introducing Node-WebKit	294
Operating HTML applications on a desktop	295
Preparing our development environment	297

Table of Contents

Installing and building our first application	299
Dissecting the package.json file	301
Building our simple application	302
Exploring our demo further	304
Dissecting our content files	304
Exploring window.js	305
Dissecting the BlueImp plugin configuration	305
Automating the creation of our project	308
Debugging your application	310
Packaging and deploying your app	310
Creating packages manually	311
Automating the process	311
Deploying your application	314
Taking things further	316
Summary	317
Chapter 13: Enhancing Performance in jQuery	319
Understanding why performance is critical	320
Monitoring the speed of jQuery using Firebug	321
Automating performance monitoring	324
Gaining insight using Google PageSpeed	330
Linting jQuery code automatically	332
Minifying code using NodeJS	335
Exploring some points of note	337
Working through a real example	338
Working out unused JavaScript	339
Implementing best practices	342
Designing a strategy for performance	347
Staying with the use of jQuery	349
Summary	350
Chapter 14: Testing jQuery	351
Revisiting QUnit	351
Installing QUnit	352
Creating a simple demo	352
Automating tests with QUnit	356
Exploring best practices when using QUnit	360
Summary	362
Index	363

Preface

Imagine a scenario, if you will, where you're an intermediate-level developer, reasonably au fait with writing code, who feels that there should be more to developing jQuery than just punching keys into a text editor.

You'd be right; anyone can write code. To take that step towards being a more rounded developer, we must think further afield. Gone are the days of writing dozens of chained statements that take a degree to understand and debug, and in their place are the decisions that help us make smarter decisions about using jQuery and that make more effective use of time in our busy lives.

As an author, I maintain that simple solutions frequently work better than complex solutions; throughout this book, we'll take look at a variety of topics that will help develop your skills, make you consider all the options, and understand that there is more to writing jQuery code.

It's going to be a great journey, with more twists and turns than a detective novel; the question is, "Are you ready?" If the answer is yes, let's make a start...

What this book covers

Chapter 1, Installing jQuery, kicks off our journey into the world of mastering jQuery, where you will learn that there is more to downloading and installing jQuery than simply using CDN or local links. We'll take a look at how to install jQuery using package managers, how we can customize the elements of our download, as well as how to add source maps and more to help fine-tune our copy of the library.

Chapter 2, Customizing jQuery, takes things further – you may find that the elements of jQuery don't quite work the way you want. In this chapter, we'll take a look at how you can create and distribute patches that can be applied temporarily in order to extend or alter the core functionality within jQuery.

Chapter 3, Organizing Your Code, explores the use of jQuery design patterns, which is a useful concept in maintaining well-organized code that makes developing and debugging easier. We'll take a look at some examples of patterns and how they fit in with jQuery.

Chapter 4, Working with Forms, takes a look at the doyen of form functionality - validating responses on forms. We'll explore how you can be more effective at form validation, before using it to great effect in a contact form that employs AJAX, and develop a file upload form.

Chapter 5, Integrating AJAX, examines how we can improve the speed of loading data on static sites, with the use of callbacks to help manage multiple AJAX requests. We'll take a look at AJAX best practices and explore how best to manage these requests through the use of jQuery's Deferreds and Promises functionalities.

Chapter 6, Animating in jQuery, takes us on a journey to discover how we can be smarter at managing animations within jQuery, and explores how best to manage the jQuery queue to prevent animation build-ups. We'll also learn how we can implement custom animations and why jQuery isn't always the right tool to use in order to move elements on a page.

Chapter 7, Advanced Event Handling, examines how many developers may simply use .on() or .off() to handle events, but you'll see that there is more to using these methods, if you really want to take advantage of jQuery. We'll create a number of custom events before we explore the use of event delegation to better manage when these event handlers are called in our code.

Chapter 8, Using jQuery Effects, continues our journey, with a quick recap on using effects in jQuery, as we explore how we can create custom effects with callbacks and learn how to better manage the queue that forms the basis of their use within jQuery.

Chapter 9, Using the Web Performance APIs, starts the second part of the book, where we explore some of the more interesting options available to us when using jQuery. In this chapter, we'll discover how to use the Page Visibility API with jQuery and see how we can use it to provide a smoother appearance, reduce resources, and still maintain complex animations on our pages. Intrigued? You will be, when you visit this chapter!

Chapter 10, Manipulating Images, illustrates how, with the use of jQuery and some reasonably simple math, we can apply all kinds of effects to images. We can perform something as simple as blurring images to creating custom effects. We'll then use some of these techniques to create a simple signature page that exports images, and apply all kinds of effects to images extracted from your own webcam.

Chapter 11, Authoring Advanced Plugins, covers one of the key topics of using jQuery: creating and distributing plugins. With more and more functionality being moved to using plugins, we'll cover some of the tips and tricks behind creating your own plugins; you'll see that there is more to it than just writing code!

Chapter 12, Using jQuery with the Node-WebKit Project, explores an interesting library that takes the best elements of Node, JavaScript/jQuery, CSS, and plain HTML and combines them into something that blurs the boundaries between desktops and the online world. We'll work through some existing online code and convert it for use as a desktop application, before packaging it and making it available for download online.

Chapter 13, Enhancing Performance in jQuery, takes you through some of the considerations, tips, and tricks that you need to use in order to optimize and enhance the performance of your code. You'll see how easy it is to get the basics from DOM inspectors, such as Firebug, right through to automating your tests with Grunt, and finally developing a strategy to keep monitoring the performance of your code.

Chapter 14, Testing jQuery, is the concluding chapter in our journey through the world of mastering jQuery, where we will take a look at testing our code using QUnit and how we can take advantage of Grunt to automate an otherwise routine but important task within the world of developing with jQuery.

What you need for this book

All you need to work through most of the examples in this book is a simple text or code editor, a copy of the jQuery library, Internet access, and a browser. I recommend that you install Sublime Text—either version 2 or 3; it works well with Node and Grunt, which we will use at various stages throughout the book.

Some of the examples make use of additional software, such as Node or Grunt—details are included within the appropriate chapter along with links to download the application from its source.

Who this book is for

The book is for frontend developers who want to do more than just write code, but who want to explore the tips and tricks that can be used to expand their skills within jQuery development. To get the most out of this book, you should have a good knowledge of HTML, CSS, and JavaScript and ideally be at an intermediate level with jQuery.

Conventions

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

Code words in text are shown as follows: "We'll start by extracting the relevant files from the code download for this book; for this demo, we'll need `clicktoggle.css`, `jquery.min.js`, and `clicktoggle.html`."

A block of code is set as follows:

```
$ (this).on("click", function() {  
    if (clicked) {  
        clicked = false;  
        return b.apply(this, arguments);  
    }  
    clicked = true;  
    return a.apply(this, arguments);  
});  
});
```

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

```
$ ('#section').hide(2000, 'swing', function() {  
    $(this).html("Animation Completed");  
});
```

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

```
npm install jquery
```

New terms and important words are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "When we view the page and select the **Images** tab, after a short delay we should see six new images."



Warnings or important notes appear in a box like this.



Tips and tricks appear like this.

Reader feedback

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

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

If there is a book that you need and would like to see us publish, please send us a note in the SUGGEST A TITLE form on www.packtpub.com or e-mail suggest@packtpub.com.

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

Customer Support

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

Downloading the example code

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

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books – maybe a mistake in the text or the code – we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us to improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/support>, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on 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 copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

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

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

Questions

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

1

Installing jQuery

Local or CDN, I wonder...? Which version...? Do I support old IE...?

Installing jQuery is a thankless task that has to be done countless times by any developer – it is easy to imagine that person asking some of the questions that start this chapter. It is easy to imagine why most people go with the option of using a **Content Delivery Network (CDN)** link, but there is more to installing jQuery than taking the easy route!

There are more options available, where we can be really specific about what we need to use – throughout this chapter, we will examine some of the options available to help develop your skills even further. We'll cover a number of topics, which include:

- Downloading and installing jQuery
- Customizing jQuery downloads
- Building from Git
- Using other sources to install jQuery
- Adding source map support
- Working with Modernizr as a fallback

Intrigued? Let's get started.

Downloading and installing jQuery

As with all projects that require the use of jQuery, we must start somewhere – no doubt you've downloaded and installed jQuery a thousand times; let's just quickly recap to bring ourselves up to speed.

If we browse to `http://www.jquery.com/download`, we can download jQuery using one of the two methods: downloading the compressed production version or the uncompressed development version. If we don't need to support old IE (IE6, 7, and 8), then we can choose the 2.x branch. If, however, you still have some diehards who can't (or don't want to) upgrade, then the 1.x branch must be used instead.

To include jQuery, we just need to add this link to our page:

```
<script src="http://code.jquery.com/jquery-X.X.X.js"></script>
```

Downloading the example code



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

Here, `X.X.X` marks the version number of jQuery or the Migrate plugin that is being used in the page.

Conventional wisdom states that the jQuery plugin (and this includes the Migrate plugin too) should be added to the `<head>` tag, although there are valid arguments to add it as the last statement before the closing `<body>` tag; placing it here may help speed up loading times to your site.

This argument is not set in stone; there may be instances where placing it in the `<head>` tag is necessary and this choice should be left to the developer's requirements. My personal preference is to place it in the `<head>` tag as it provides a clean separation of the script (and the CSS) code from the main markup in the body of the page, particularly on lighter sites.

I have even seen some developers argue that there is little *perceived* difference if jQuery is added at the top, rather than at the bottom; some systems, such as WordPress, include jQuery in the `<head>` section too, so either will work. The key here though is if you are perceiving slowness, then move your scripts to just before the `<body>` tag, which is considered a better practice.

Using jQuery in a development capacity

A useful point to note at this stage is that best practice recommends that CDN links should not be used within a development capacity; instead, the uncompressed files should be downloaded and referenced locally. Once the site is complete and is ready to be uploaded, then CDN links can be used.

Adding the jQuery Migrate plugin

If you've used any version of jQuery prior to 1.9, then it is worth adding the jQuery Migrate plugin to your pages. The jQuery Core team made some significant changes to jQuery from this version; the Migrate plugin will temporarily restore the functionality until such time that the old code can be updated or replaced.

The plugin adds three properties and a method to the jQuery object, which we can use to control its behavior:

Property or Method	Comments
<code>jQuery.migrateWarnings</code>	This is an array of string warning messages that have been generated by the code on the page, in the order in which they were generated. Messages appear in the array only once even if the condition has occurred multiple times, unless <code>jQuery.migrateReset()</code> is called.
<code>jQuery.migrateMute</code>	Set this property to <code>true</code> in order to prevent console warnings from being generated in the debugging version. If this property is set, the <code>jQuery.migrateWarnings</code> array is still maintained, which allows programmatic inspection without console output.
<code>jQuery.migrateTrace</code>	Set this property to <code>false</code> if you want warnings but don't want traces to appear on the console.
<code>jQuery.migrateReset()</code>	This method clears the <code>jQuery.migrateWarnings</code> array and "forgets" the list of messages that have been seen already.

Adding the plugin is equally simple—all you need to do is add a link similar to this, where X represents the version number of the plugin that is used:

```
<script src="http://code.jquery.com/jquery-migrate-
X.X.X.js"></script>
```

If you want to learn more about the plugin and obtain the source code, then it is available for download from <https://github.com/jquery/jquery-migrate>.

Using a CDN

We can equally use a CDN link to provide our jQuery library – the principal link is provided by **MaxCDN** for the jQuery team, with the current version available at <http://code.jquery.com>. We can, of course, use CDN links from some alternative sources, if preferred – a reminder of these is as follows:

- Google (<https://developers.google.com/speed/libraries/devguide#jquery>)
- Microsoft (http://www.asp.net/ajaxlibrary/cdn.ashx#jQuery_Releases_on_the_CDN_0)
- CDNJS (<http://cdnjs.com/libraries/jquery/>)
- jsDelivr (<http://www.jsdelivr.com/#%!jquery>)

Don't forget though that if you need, we can always save a copy of the file provided on CDN locally and reference this instead. The jQuery CDN will always have the latest version, although it may take a couple of days for updates to appear via the other links.

Using other sources to install jQuery

Right. Okay, let's move on and develop some code! "What's next?" I hear you ask.

Aha! If you thought downloading and installing jQuery from the main site was the only way to do this, then you are wrong! After all, this book is about mastering jQuery, so you didn't think I will only talk about something that I am sure you are already familiar with, right?

Yes, there are more options available to us to install jQuery than simply using the CDN or main download page. Let's begin by taking a look at using Node.



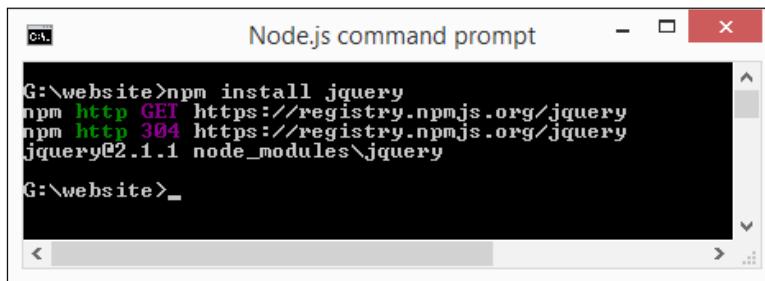
Each demo is based on Windows, as this is the author's preferred platform; alternatives are given, where possible, for other platforms.

Using NodeJS to install jQuery

So far, we've seen how to download and reference jQuery, which is to use the download from the main jQuery site or via a CDN. The downside of this method is the manual work required to keep our versions of jQuery up to date! Instead, we can use a package manager to help manage our assets. Node.js is one such system. Let's take a look at the steps that need to be performed in order to get jQuery installed:

1. We first need to install Node.js – head over to <http://www.nodejs.org> in order to download the package for your chosen platform; accept all the defaults when working through the wizard (for Mac and PC).
2. Next, fire up a Node command prompt and then change to your project folder.
3. In the prompt, enter this command:

```
npm install jquery
```
4. Node will fetch and install jQuery – it displays a confirmation message when the installation is complete:

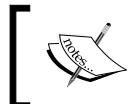


The screenshot shows a Windows command prompt window titled "Node.js command prompt". The command entered was "G:\website>npm install jquery". The output shows the process of fetching the package from the npm registry and installing it into the node_modules directory. The final line shows the package version installed: "jquery@2.1.1 node_modules\jquery".

5. You can then reference jQuery by using this link:

`<name of drive>:\website\node_modules\jquery\dist\jquery.min.js.`

Node is now installed and ready for use – although we've installed it in a folder locally, in reality, we will most likely install it within a subfolder of our local web server. For example, if we're running WampServer, we can install it, then copy it into the /wamp/www/js folder, and reference it using `http://localhost/js/jquery.min.js`.



If you want to take a look at the source of the jQuery **Node Package Manager (NPM)** package, then check out <https://www.npmjs.org/package/jquery>.

Using Node to install jQuery makes our work simpler, but at a cost. Node.js (and its package manager, NPM) is primarily aimed at installing and managing JavaScript components and expects packages to follow the **CommonJS** standard. The downside of this is that there is no scope to manage any of the other assets that are often used within websites, such as fonts, images, CSS files, or even HTML pages.

"Why will this be an issue?," I hear you ask. Simple, why make life hard for ourselves when we can manage all of these assets automatically and still use Node?

Installing jQuery using Bower

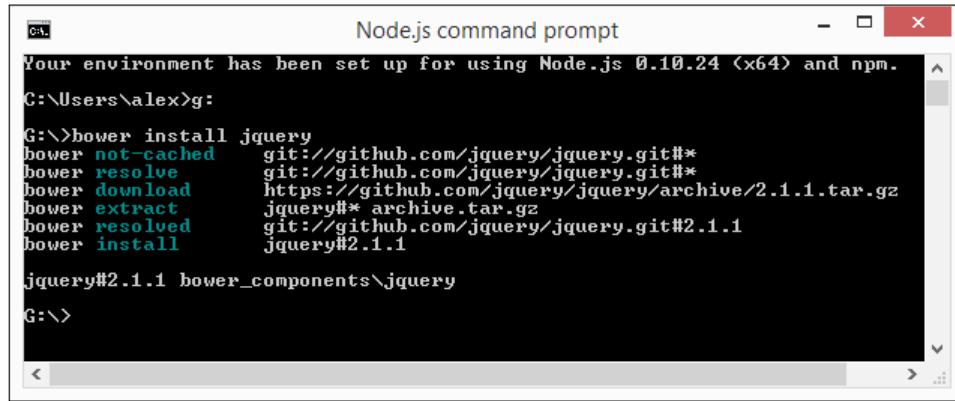
A relatively new addition to the library is the support for installation using Bower—based on Node, it's a package manager that takes care of the fetching and installing of packages from over the Internet. It is designed to be far more flexible about managing the handling of multiple types of assets (such as images, fonts, and CSS files) and does not interfere with how these components are used within a page (unlike Node).

For the purpose of this demo, I will assume that you have already installed it from the previous section; if not, you will need to revisit it before continuing with the following steps:

1. Bring up the Node command prompt, change to the drive where you want to install jQuery, and enter this command:

```
bower install jquery
```

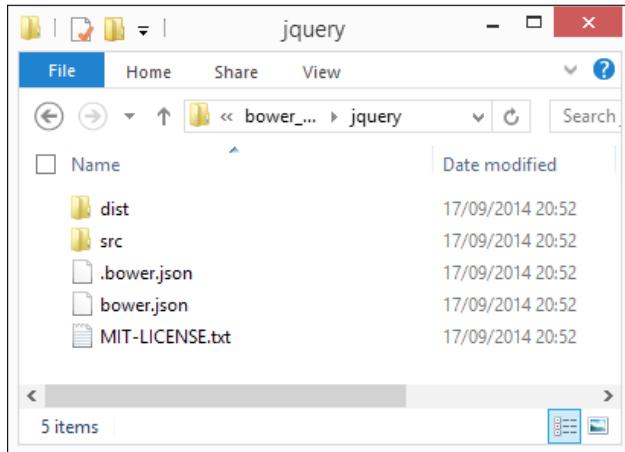
This will download and install the script, displaying the confirmation of the version installed when it has completed, as shown in the following screenshot:



The screenshot shows a Windows command prompt window titled "Node.js command prompt". The window displays the following text:

```
Your environment has been set up for using Node.js 0.10.24 <x64> and npm.
C:\Users\alex>g:
G:\>bower install jquery
bower not-cached  git://github.com/jquery/jquery.git#*
bower resolve   git://github.com/jquery/jquery.git#*
bower download  https://github.com/jquery/jquery/archive/2.1.1.tar.gz
bower extract   jquery##* archive.tar.gz
bower resolved   git://github.com/jquery/jquery.git#2.1.1
bower install    jquery#2.1.1
jquery#2.1.1 bower_components\jquery
G:\>
```

The library is installed in the `bower_components` folder on your PC. It will look similar to this example, where I've navigated to the `jquery` subfolder underneath:



By default, Bower will install jQuery in its `bower_components` folder. Within `bower_components/jquery/dist/`, we will find an uncompressed version, compressed release, and source map file. We can then reference jQuery in our script using this line:

```
<script src="/bower_components/jquery/jquery.js"></script>
```

We can take this further though. If we don't want to install the extra files that come with a Bower installation by default, we can simply enter this in a command prompt instead to just install the minified version 2.1 of jQuery:

```
bower install http://code.jquery.com/jquery-2.1.0.min.js
```

Now, we can be really clever at this point; as Bower uses Node's JSON files to control what should be installed, we can use this to be really selective and set Bower to install additional components at the same time. Let's take a look and see how this will work—in the following example, we'll use Bower to install jQuery 2.1 and 1.10 (the latter to provide support for IE6-8):

1. In the Node command prompt, enter the following command:

```
bower init
```

This will prompt you for answers to a series of questions, at which point you can either fill out information or press *Enter* to accept the defaults.

Installing jQuery

2. Look in the project folder; you should find a `bower.json` file within. Open it in your favorite text editor and then alter the code as shown here:

```
{  
  "ignore": [ "**/*", "node_modules", "bower_components",  
  "test", "tests" ],  
  "dependencies": {  
    "jquery-legacy": "jquery#1.11.1",  
    "jquery-modern": "jquery#2.10"  
  }  
}
```

At this point, you have a `bower.json` file that is ready for use. Bower is built on top of Git, so in order to install jQuery using your file, you will normally need to publish it to the Bower repository.

Instead, you can install an additional Bower package, which will allow you to install your custom package without the need to publish it to the Bower repository:

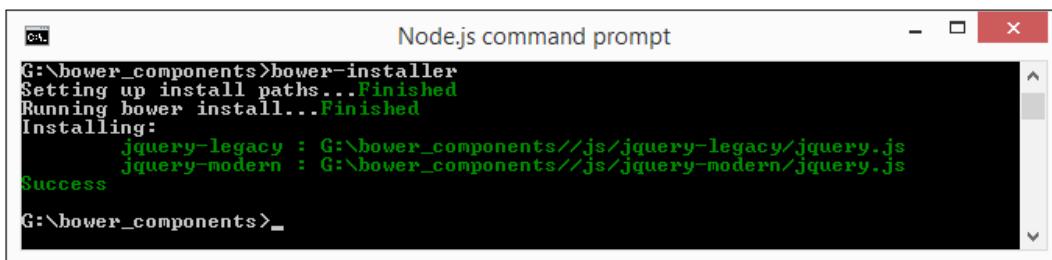
1. In the Node command prompt window, enter the following at the prompt:

```
npm install -g bower-installer
```

2. When the installation is complete, change to your project folder and then enter this command line:

```
bower-installer
```

3. The `bower-installer` command will now download and install both the versions of jQuery, as shown here:



The screenshot shows a terminal window titled "Node.js command prompt". The command entered was "G:\bower_components>bower-installer". The output indicates that paths were set up and the Bower install process was completed successfully. It shows the installation of two packages: "jquery-legacy" and "jquery-modern", both pointing to "G:\bower_components\js\jquery-legacy\jquery.js" and "G:\bower_components\js\jquery-modern\jquery.js" respectively. The word "Success" is printed at the end of the output.

```
Node.js command prompt  
G:\bower_components>bower-installer  
Setting up install paths...Finished  
Running bower install...Finished  
Installing:  
  jquery-legacy : G:\bower_components\js\jquery-legacy\jquery.js  
  jquery-modern : G:\bower_components\js\jquery-modern\jquery.js  
Success  
G:\bower_components>_
```

At this stage, you now have jQuery installed using Bower. You're free to upgrade or remove jQuery using the normal Bower process at some point in the future.



If you want to learn more about how to use Bower, there are plenty of references online; <https://www.openshift.com/blogs/day-1-bower-manage-your-client-side-dependencies> is a good example of a tutorial that will help you get accustomed to using Bower. In addition, there is a useful article that discusses both Bower and Node, available at <http://tech.pro/tutorial/1190/package-managers-an-introductory-guide-for-the-uninitiated-front-end-developer>.

Bower isn't the only way to install jQuery though—while we can use it to install multiple versions of jQuery, for example, we're still limited to installing the entire jQuery library.

We can improve on this by referencing only the elements we need within the library. Thanks to some extensive work undertaken by the jQuery Core team, we can use the **Asynchronous Module Definition (AMD)** approach to reference only those modules that are needed within our website or online application.

Using the AMD approach to load jQuery

In most instances, when using jQuery, developers are likely to simply include a reference to the main library in their code. There is nothing wrong with it per se, but it loads a lot of extra code that is surplus to our requirements.

A more efficient method, although one that takes a little effort in getting used to, is to use the AMD approach. In a nutshell, the jQuery team has made the library more modular; this allows you to use a loader such as require.js to load individual modules when needed.

It's not suitable for every approach, particularly if you are a heavy user of different parts of the library. However, for those instances where you only need a limited number of modules, then this is a perfect route to take. Let's work through a simple example to see what it looks like in practice.



Before we start, we need one additional item—the code uses the Fira Sans regular custom font, which is available from Font Squirrel at <http://www.fontsquirrel.com/fonts/fira-sans>.

Let's make a start using the following steps:

1. The Fira Sans font doesn't come with a web format by default, so we need to convert the font to use the web font format. Go ahead and upload the `FiraSans-Regular.woff` file to Font Squirrel's web font generator at <http://www.fontsquirrel.com/tools/webfont-generator>. When prompted, save the converted file to your project folder in a subfolder called `fonts`.

2. We need to install jQuery and RequireJS into our project folder, so fire up a Node.js command prompt and change to the project folder.
3. Next, enter these commands one by one, pressing *Enter* after each:

```
bower install jquery  
bower install requirejs
```

4. We need to extract a copy of the `amd.html` and `amd.css` files from the code download link that accompanies this book—it contains some simple markup along with a link to `require.js`; the `amd.css` file contains some basic styling that we will use in our demo.
5. We now need to add in this code block, immediately below the link for `require.js`—this handles the calls to jQuery and RequireJS, where we're calling in both jQuery and Sizzle, the selector engine for jQuery:

```
<script>  
    require.config({  
        paths: {  
            "jquery": "bower_components/jquery/src",  
            "sizzle": "bower_components/jquery/src/sizzle/dist/sizzle"  
        }  
    });  
    require(["js/app"]);  
</script>
```

6. Now that jQuery has been defined, we need to call in the relevant modules. In a new file, go ahead and add the following code, saving it as `app.js` in a subfolder marked `js` within our project folder:

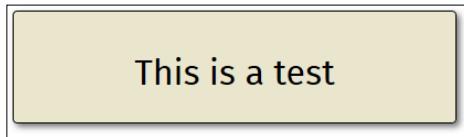
```
define(["jquery/core/init", "jquery/attributes/classes"],  
function($) {  
    $("div").addClass("decoration");  
});
```



We used `app.js` as the filename to tie in with the `require(["js/app"]);` reference in the code.



7. If all went well, when previewing the results of our work in a browser, we'll see this message:



Although we've only worked with a simple example here, it's enough to demonstrate how easy it is to only call those modules we need to use in our code rather than call the entire jQuery library. True, we still have to provide a link to the library, but this is only to tell our code where to find it; our module code weighs in at 29 KB (10 KB when gzipped), against 242 KB for the uncompressed version of the full library!



There is a completed version of our code available in the code download link that accompanies this book—look for and run the `amd-finished.html` file to view the results.



Now, there may be instances where simply referencing modules using this method isn't the right approach; this may apply if you need to reference lots of different modules regularly.

A better alternative is to build a custom version of the jQuery library that only contains the modules that we need to use and the rest are removed during build. It's a little more involved but worth the effort—let's take a look at what is involved in the process.

Customizing the downloads of jQuery from Git

If we feel so inclined, we can really push the boat out and build a custom version of jQuery using the JavaScript task runner, Grunt. The process is relatively straightforward but involves a few steps; it will certainly help if you have some prior familiarity with Git!



The demo assumes that you have already installed Node.js—if you haven't, then you will need to do this first before continuing with the exercise.



Installing jQuery

Okay, let's make a start by performing the following steps:

1. You first need to install Grunt if it isn't already present on your system—bring up the Node.js command prompt and enter this command:

```
npm install -g grunt-cli
```

2. Next, install Git—for this, browse to <http://msysgit.github.io/> in order to download the package.
3. Double-click on the setup file to launch the wizard, accepting all the defaults is sufficient for our needs.

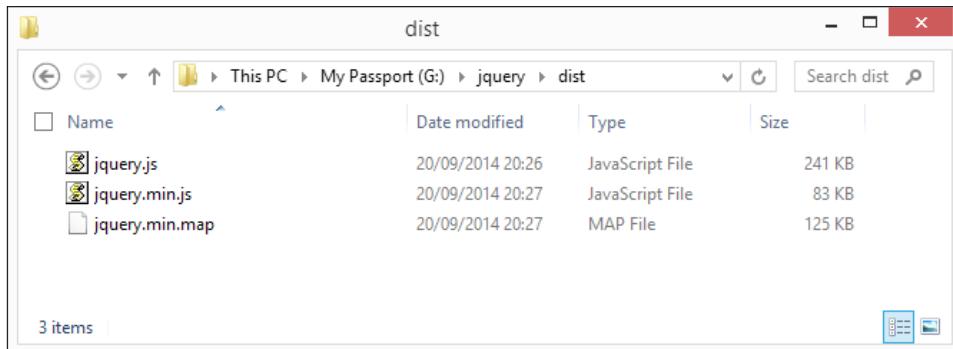


If you want more information on how to install Git, head over and take a look at <https://github.com/msysgit/msysgit/wiki/InstallMSysGit> for more details.

4. Once Git is installed, change to the `jquery` folder from within the command prompt and enter this command to download and install the dependencies needed to build jQuery:

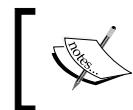
```
npm install
```

5. The final stage of the build process is to build the library into the file we all know and love; from the same command prompt, enter this command:
`grunt`
6. Browse to the `jquery` folder—within this will be a folder called `dist`, which contains our custom build of jQuery, ready for use, as shown in the following screenshot:



Removing redundant modules

If there are modules within the library that we don't need, we can run a custom build. We can set the Grunt task to remove these when building the library, leaving in those that are needed for our project.



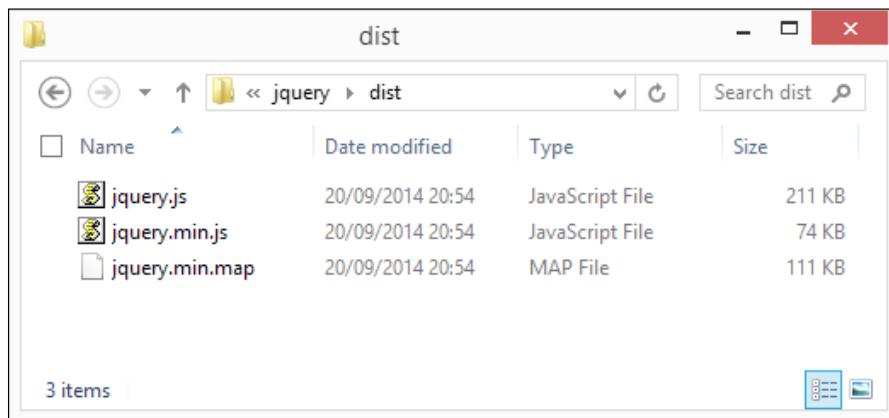
For a complete list of all the modules that we can exclude, see <https://github.com/jquery/jquery#modules>.



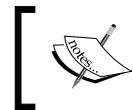
For example, to remove AJAX support from our build, we can run this command in place of step 5, as shown previously:

```
grunt custom:-ajax
```

This results in a file saving on the original raw version of 30 KB as shown in the following screenshot:



The JavaScript and map files can now be incorporated into our projects in the usual way.



For a detailed tutorial on the build process, this article by Dan Wellman is worth a read (<https://www.packtpub.com/books/content/building-custom-version-jquery>).



Using a GUI as an alternative

There is an online GUI available, which performs much the same tasks, without the need to install Git or Grunt. It's available at <http://projects.jga.me/jquery-builder/>, although it is worth noting that it hasn't been updated for a while!

Okay, so we have jQuery installed; let's take a look at one more useful function that will help in the event of debugging errors in our code. Support for source maps has been made available within jQuery since version 1.9. Let's take a look at how they work and see a simple example in action.

Adding source map support

Imagine a scenario, if you will, where you've created a killer site, which is running well, until you start getting complaints about problems with some of the jQuery-based functionality that is used on the site. Sounds familiar?

Using an uncompressed version of jQuery on a production site is not an option; instead we can use source maps. Simply put, these map a compressed version of jQuery against the relevant line in the original source.

Historically, source maps have given developers a lot of heartache when implementing, to the extent that the jQuery Team had to revert to disabling the automatic use of maps!



For best effects, it is recommended that you use a local web server, such as WAMP (PC) or MAMP (Mac), to view this demo and that you use Chrome as your browser.

Source maps are not difficult to implement; let's run through how you can implement them:

1. From the code download link that accompanies this book, extract a copy of the `sourcemap` folder and save it to your project area locally.
2. Press `Ctrl + Shift + I` to bring up the **Developer Tools** in Chrome.

- Click on **Sources**, then double-click on the `sourcemap.html` file—in the code window, and finally click on **17**, as shown in the following screenshot:

The screenshot shows the Google Chrome Developer Tools interface. The title bar says "Developer Tools - http://localhost/sourcemap/sourcemap.html". The tabs at the top are Elements, Network, Sources, Timeline, Profiles, Resources, Audits, and Console. The Sources tab is active. In the left sidebar, under "localhost" and "sourcemap", there is a "js" folder containing "sourcemap.html". The main code editor window shows the following JavaScript code:

```

15 $(document).ready(function(){
16   $("button").click(function(){
17     $("#block").animate({
18       left:'250px'
19     }, 1500);
20   });
21 });
22 </script>
23 </body>
24 </html>

```

Line 17 is highlighted in blue. The right-hand pane shows the Call Stack and Scope Variables sections.

- Now, run the demo in Chrome—we will see it paused; revert back to the developer toolbar where line **17** is highlighted. The relevant calls to the jQuery library are shown on the right-hand side of the screen:

The screenshot shows the Google Chrome Developer Tools interface. The title bar says "Developer Tools - http://localhost/sourcemap/sourcemap.html". The tabs at the top are Elements, Network, Sources, Timeline, Profiles, Resources, Audits, and Console. The Sources tab is active. In the left sidebar, under "localhost" and "sourcemap", there is a "js" folder containing "sourcemap.html". The main code editor window shows the same JavaScript code as before, with line 17 highlighted in blue. The right-hand pane shows the Call Stack and Scope Variables sections. The "Call Stack" section highlights the entry `n.event.dispatch`.

- If we double-click on the `n.event.dispatch` entry on the right, Chrome refreshes the toolbar and displays the original source line (highlighted) from the jQuery library, as shown here:

The screenshot shows the Google Chrome Developer Tools interface. The title bar says "Developer Tools - http://localhost/sourcemap/sourcemap.html". The tabs at the top are Elements, Network, Sources, Timeline, Profiles, Resources, Audits, and Console. The Sources tab is active. In the left sidebar, under "localhost" and "sourcemap", there is a "js" folder containing "sourcemap.html". The main code editor window shows the following JavaScript code:

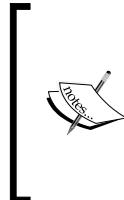
```

4404 event.handleObj = handleObj;
4405 event.data = handleObj.data;
4406
4407 ret = ( (jQuery.event.special[ handleObj.type ] || {})[ event.type ] ||
4408   event.special[ event.type ] ).apply( matched.elem, args );
4409 if ( ret !== undefined ) {
4410   if ( (event.result = ret) === false )
4411     event.preventDefault();
4412
4413   if ( event.stopPropagation )
4414     event.stopPropagation();

```

Line 4409 is highlighted in blue. The right-hand pane shows the Call Stack and Scope Variables sections. The "Call Stack" section highlights the entry `n.event.dispatch`.

It is well worth spending the time to get to know source maps—all the latest browsers support it, including IE11. Even though we've only used a simple example here, it doesn't matter as the principle is exactly the same, no matter how much code is used in the site.



For a more in-depth tutorial that covers all the browsers, it is worth heading over to <http://blogs.msdn.com/b/davrous/archive/2014/08/22/enhance-your-javascript-debugging-life-thanks-to-the-source-map-support-available-in-ie11-chrome-opera-and-firefox.aspx>—it is worth a read!

Adding support for source maps

In the previous section, where we've just previewed the source map, source map support has already been added to the library. It is worth noting though that source maps are not included with the current versions of jQuery by default. If you need to download a more recent version or add support for the first time, then follow these steps:

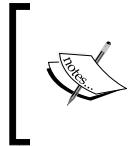
1. Source maps can be downloaded from the main site using `http://code.jquery.com/jquery-X.X.X.min.map`, where X represents the version number of jQuery being used.
2. Open a copy of the minified version of the library and then add this line at the end of the file:
`//# sourceMappingURL=jquery.min.map`
3. Save it and then store it in the JavaScript folder of your project. Make sure you have copies of both the compressed and uncompressed versions of the library within the same folder.

Let's move on and look at one more critical part of loading jQuery: if, for some unknown reason, jQuery becomes completely unavailable, then we can add a fallback position to our site that allows graceful degradation. It's a small but crucial part of any site and presents a better user experience than your site simply falling over!

Working with Modernizr as a fallback

A best practice when working with jQuery is to ensure that a fallback is provided for the library, should the primary version not be available. (Yes, it's irritating when it happens, but it can happen!)

Typically, we might use a little JavaScript, such as the following example, in the best practice suggestions. This would work perfectly well but doesn't provide a graceful fallback. Instead, we can use Modernizr to perform the check for us and provide a graceful degradation if all fails.



Modernizr is a feature detection library for HTML5/CSS3, which can be used to provide a standardized fallback mechanism in the event of a functionality not being available. You can learn more at <http://www.modernizr.com>.

As an example, the code might look like this at the end of our website page. We first try to load jQuery using the CDN link, falling back to a local copy if that hasn't worked or an alternative if both fail:

```
<body>
<script src="js/modernizr.js"></script>
<script type="text/javascript">
    Modernizr.load([
        {
            load: 'http://code.jquery.com/jquery-2.1.1.min.js',
            complete: function () {
                // Confirm if jQuery was loaded using CDN link
                // if not, fall back to local version
                if ( !window.jQuery ) {
                    Modernizr.load('js/jquery-latest.min.js');
                }
            }
        },
        // This script would wait until fallback is loaded, before
        loading
        { load: 'jquery-example.js' }
    ]);
</script>
</body>
```

In this way, we can ensure that jQuery either loads locally or from the CDN link—if all else fails, then we can at least make a graceful exit.

Best practices for loading jQuery

So far, we've examined several ways of loading jQuery into our pages, over and above the usual route of downloading the library locally or using a CDN link in our code. Now that we have it installed, it's a good opportunity to cover some of the best practices we should try to incorporate into our pages when loading jQuery:

- Always try to use a CDN to include jQuery on your production site. We can take advantage of the high availability and low latency offered by CDN services; the library may already be precached too, avoiding the need to download it again.
- Try to implement a fallback on your locally hosted library of the same version. If CDN links become unavailable (and they are not 100 percent infallible), then the local version will kick in automatically, until the CDN link becomes available again:

```
<script type="text/javascript" src="//code.jquery.com/jquery-1.11.1.min.js"></script>
<script>window.jQuery || document.write('<script
src="js/jquery-1.11.1.min.js"></script>')</script>
```
- Note that although this will work equally well as using Modernizr, it doesn't provide a graceful fallback if both the versions of jQuery should become unavailable. Although one hopes to never be in this position, at least we can use CSS to provide a graceful exit!
- Use protocol-relative/protocol-independent URLs; the browser will automatically determine which protocol to use. If HTTPS is not available, then it will fall back to HTTP. If you look carefully at the code in the previous point, it shows a perfect example of a protocol-independent URL, with the call to jQuery from the main jQuery Core site.
- If possible, keep all your JavaScript and jQuery inclusions at the bottom of your page—scripts block the rendering of the rest of the page until they have been fully rendered.
- Use the jQuery 2.x branch, unless you need to support IE6-8; in this case, use jQuery 1.x instead—do not load multiple jQuery versions.
- If you load jQuery using a CDN link, always specify the complete version number you want to load, such as `jquery-1.11.1.min.js`.
- If you are using other libraries, such as Prototype, MooTools, Zepto, and so on, that use the `$` sign as well, try not to use `$` to call jQuery functions and simply use jQuery instead. You can return the control of `$` back to the other library with a call to the `$.noConflict()` function.
- For advanced browser feature detection, use Modernizr.

It is worth noting that there may be instances where it isn't always possible to follow best practices; circumstances may dictate that we need to make allowances for requirements, where best practices can't be used. However, this should be kept to a minimum where possible; one might argue that there are flaws in our design if most of the code doesn't follow best practices!

Summary

If you thought that the only methods to include jQuery were via a manual download or using a CDN link, then hopefully this chapter has opened your eyes to some alternatives – let's take a moment to recap what we have learned.

We kicked off with a customary look at how most developers are likely to include jQuery before quickly moving on to look at other sources.

We started with a look at how to use Node, before turning our attention to using the Bower package manager. Next, we had a look at how we can reference individual modules within jQuery using the AMD approach. We then moved on and turned our attention to creating custom builds of the library using Git. We then covered how we can use source maps to debug our code, with a look at enabling support for them within Google's Chrome browser.

To round out our journey of loading jQuery, we saw what might happen if we can't load jQuery at all and how we can get around this, by using Modernizr to allow our pages to degrade gracefully. We then finished the chapter with some of the best practices that we can follow when referencing jQuery.

In the next chapter, we'll kick things into a gear by taking a look at how we can customize jQuery. This can be done by replacing or modifying a function or applying a patch during runtime; are you ready to get stuck in?

2

Customizing jQuery

Okay, so we've downloaded a version of jQuery...what do we do next, I wonder?

This is a really good question – let me reveal all!

jQuery has, over the years, become an accomplished library and is used in millions of websites around the world. While we can usually find a way to fulfill a need using the library, there may be instances where we have to provide our own patch or alteration, to satisfy our needs.

We can use a plugin, but that gets tedious after a while – it soon becomes a case of "plugin this, plugin that" syndrome, where we become too reliant on plugins. Instead, we can look to the override functionality within jQuery itself; yes, it has some risks, but as we'll see, it is well worth the effort. Throughout this chapter, we'll cover the basics of overriding jQuery, some of the benefits and pitfalls of doing so, and work our way through some examples of replacing the functionality. We will cover the following topics:

- Introducing duck punching
- Replacing or modifying existing behaviors
- Creating a basic monkey patch
- Considering the benefits and pitfalls of monkey patching
- Distributing or applying patches

Ready to begin your adventure...? Let's get started!

Getting prepared

At this point, I will recommend that you create a project folder somewhere on your PC—for the purposes of this demo, I will assume that it is called `project` and is at the root of your main hard disk or `C:` drive.

Within the folder, go ahead and create several subfolders; these need to be called `fonts`, `css`, `js`, and `img`.

Patching the library on the run

Over the years, hundreds of developers have spent countless hours creating patches for jQuery, to either fix a bug of some description or provide new functionality within the library.

The usual route is to submit a pull request against the Core jQuery library for peer consideration. As long as the patch works as expected and does not cause issues elsewhere in the library, then it will be submitted to core.

The downside of this approach means that we're constrained by the release schedule for jQuery; while the developers do an outstanding job, it nevertheless can take time before a patch is committed to core.

Introducing monkey patching

What to do? Do we wait in the hope that our patch will be committed?

For some, this won't be an issue—for others, patience may not be their strongest virtue and waiting is the last thing they will want to do! Fortunately, we can get around this by using a method called monkey patching.

Now—before you ask—let me tell you that I'm not advocating any form of animal cruelty! **Monkey patching**, or **duck punching** as it is otherwise known, is a valid technique to create a patch that temporarily overrides the existing functionality within the jQuery Core library during runtime. Monkey patching comes with its risks: the primary one being that of clashing, should an update introduce a method or function of the same name within the library.



Later in this chapter, we'll take a look at some of the risks that need to be considered.



That said, if monkey patching is used with care and forethought, it can be used to update functionality until a more permanent fix can be applied. It's time, I think, for a demo—we'll be taking a look at how we can improve animation support in jQuery, but first let's take a look at the basics of replacing or modifying the jQuery core at runtime.

Replacing or modifying existing behaviors

So, how can we effect a (temporary) change in the core functionality of jQuery?

It all starts with the use of an **Immediately Invoked Function Expression (IIFE)**; we then simply save a version of the original function before overriding it with our new function.



You may have heard the term *self-executing anonymous function* being used; it is a misleading phrase, although it means the same thing as an IIFE, which is a more accurate description.



Let's see what the basic framework looks like in action:

```
(function ($) {
    // store original reference to the method
    var _old = $.fn.method;
    $.fn.method = function (arg1, arg2) {
        if (... condition ...) {
            return ....
        }
        else { // do the default
            return _old.apply(this, arguments);
        }
    };
}) (jQuery);
```

If you were expecting something more complex, then I am sorry to disappoint you; there isn't a great deal of complexity required for a basic monkey patch! The extent of what goes into a patch will really come down to what it is that you are trying to fix or alter within the existing code.

To prove that this really is all that is required, let's take a look at an (albeit over-simplified) example. In the example, we'll use a standard click handler to show the response that a dog will give to its owner...except that our dog seems to have developed a personality problem.

Creating a basic monkey patch

"A personality change?" I hear you ask. Yes, that's right; our dog seems to like miaowing.... (I can't think of a reason why; I don't know of any that would!)

In our example, we're going to use a simple click handler to prove that (in some cases) our dog can meow; we will then work through what is required in order to persuade it to do what it should do.

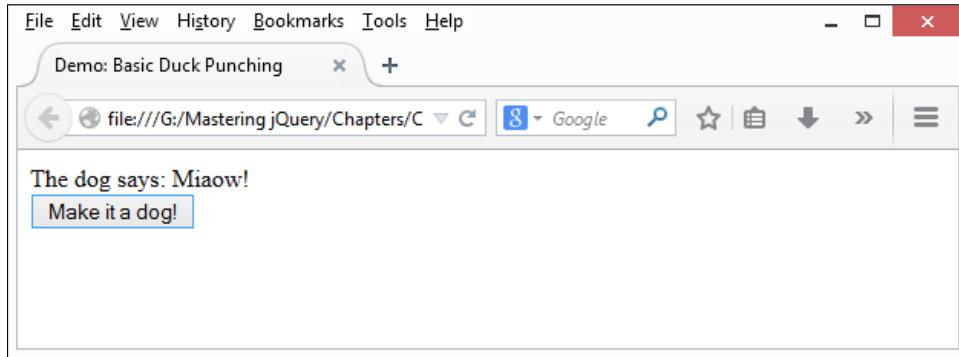
1. Let's start by cracking open a text editor of our choice and then adding the following markup as a basis for our patch:

```
<!DOCTYPE html>
<head>
    <title>Demo: Basic Duck Punching</title>
    <meta charset="utf-8">
    <script src="js/jquery.min.js"></script>
    <script src="js/duck.js"></script>
</head>
<body>
    <div>Hello World</div>
    <button>Make it a dog!</button>
</body>
</html>
```

2. Save it as the `duck.html` file. In a separate file, we need to animate our button, so let's first add in a simple event handler for this purpose:

```
$(document).ready(function() {
    jQuery.fn.toBark = function() {
        this.text("The dog says: Miaow!")
    };
    $('button').on('click', function() {
        $('div').toBark();
    });
})
```

At this point, if we run the demo in a browser and then click on **Make it a dog!**, we can definitely see that our poor pet has some issues, as shown in the following screenshot:

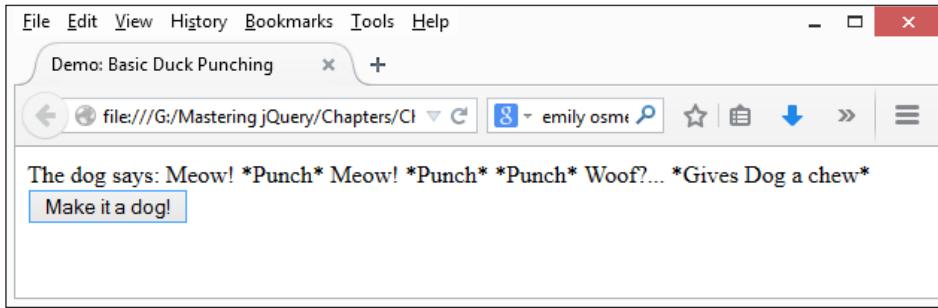


We clearly need to show it the error of its ways, so let's fix that now.

3. To fix the problem, we need to override the original `toBark()` function. With our new fixed replacement; this will take the form of a monkey patch. Insert the following code immediately below the `.on()` click handler, leaving a line space for clarity:

```
(function($) {
    var orig = $.fn.toBark;
    $.fn.toBark = function() {
        orig.apply(this, arguments);
        if (this.text() === 'The dog says: Miaow!') {
            this.append(" *Punch* Miaow! *Punch* *Punch*
Woof?... *Gives Dog a chew*");
        }
    };
} (jQuery));
```

4. If all is well, we should now at least see that our dog has come to its senses, albeit gradually, as shown in the following screenshot:



This little exercise, albeit heavily simplified, illustrates some key points—it's worth spending a little time going through this in more detail, so let's do that now.

Dissecting our monkey patch

The patching of a core library should be done with care and consideration; the technical process may be straightforward, but it will raise some questions that need to be answered first. We will cover some of these questions later in the chapter, but for now, let's assume that we need to apply a patch.

The basic patch takes the format of an IIFE—we include all the functionality within a single module; its scope is protected from the environment in which it is placed.



For a more detailed explanation of IIFEs, please refer to http://en.wikipedia.org/wiki/Immediately-invoked_function_expression.



In our example, we started by storing a copy of the original function as an object, within `orig`. We then initiated our new replacement, the `.toBark()` function, within which we first call our `.toBark()` function, but follow it with the replacement:

```
function($) {
    var orig = $.fn.toBark;
    $.fn.toBark = function() {
        orig.apply(this, arguments);
        if (this.text() === 'The dog says: Miaow!') {
            this.append(" *Punch* Miaow! *Punch* *Punch*
Woof?... *Gives Dog a chew*");
        }
    };
}
```

```

        }
    };
} (jQuery));

```

A key part of our patch is the use of the `.apply()` function – this will call a function with the context being set to the object where the function is applied. In this instance, within the function, referencing the `this` keyword will refer to that object.

Using an IIFE in the format used in our demo presents a number of advantages, as follows:

- We can reduce the scope's lookup – IIFEs allow you to pass commonly used objects to the anonymous function, so they can be referenced within the IIFE at a local scope



As JavaScript first looks for properties within the local scope, this removes the need to look globally, providing faster lookup speeds and performance. Using an IIFE prevents the local variable from being overwritten by a global variable.

- IIFEs help optimize the code through minification – we can pass objects to an IIFE as local values; a minifier can reduce the names of each global object to a single letter, provided there isn't a variable already present with the same name

The downside of using an IIFE is readability; if our IIFE contains a lot of code, then we have to scroll to the top in order to work out what objects are being passed. In more complex examples, we can consider using a pattern developed by Greg Franko in order to get around this issue:

```

(function (library) {
    // Call the second IIFE and locally pass in the global jQuery,
    window, and document objects
    library(window, document, window.jQuery);
}
// Locally scoped parameters
(function (window, document, $) {
    // Library code goes here
}));

```

It's important to note that this pattern is about splitting the variables into two sections so that we can avoid the need to scroll up and down the page too often; it will still produce the same end result.

We will delve more into using patterns within jQuery, in *Chapter 3, Organizing Your Code*. Now that we've seen a patch in action, let's move on and take some time to consider some of the benefits we may gain from using the monkey patching process.

Considering the benefits of monkey patching

Okay, so we've seen what a typical patch will look like; the question, though, is, why would we want to patch the core library functionality using this method?

This is a very good question—it's a method that has its risks (as we will see later in this chapter, in the *Considering the pitfalls of monkey patching* section). The key to using this method is to take a considered approach; with this in mind, let's take a moment to consider the benefits of duck punching jQuery:

- We can replace methods, attributes, or functions at runtime, where they lack functionality or contain a bug that needs to be fixed and we can't wait for an official patch
- Duck punching jQuery allows you to modify or extend the existing behavior of jQuery without maintaining a private copy of the source code
- We have the safety net of being able to apply a patch to objects running in memory, instead of the source code; in other words, if it goes completely wrong, we can simply pull the patch from the site and leave the original source code untouched
- Monkey patching is a good way to distribute security or behavioral fixes that live alongside the original source code; if there is any doubt with the resiliency of a patch, we can stress test it before committing it to the source

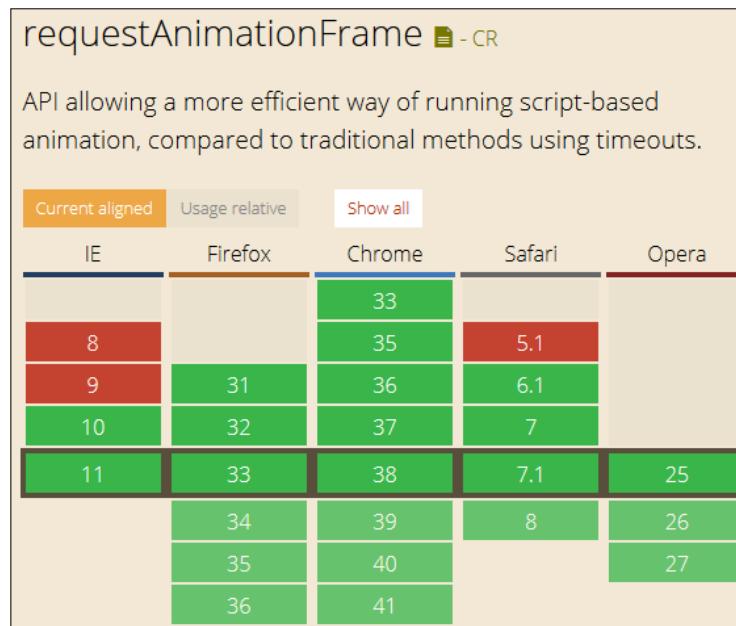
Enough of the talking, let's get to coding some demos! We're going to work through some example patches that can be equally applied to jQuery, beginning with a look at animation.

Updating animation support in jQuery

If you have spent any time developing with jQuery, you've most likely created some form of animation that included managing changes at a regular frequency – does this sound familiar?

We can, of course, use the `setInterval()` function for this, but it – like the `setTimeout()` function – is not ideal. Both these functions have a delay before being initiated, which varies from browser to browser; they are both equally resource intensive!

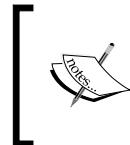
Instead, we can use the **requestAnimationFrame (rAF)** API, which is now supported by most modern browsers, according to this chart from caniuse.com where green labels show which browser versions support **requestAnimationFrame**:



The great thing about the `requestAnimationFrame` API is that it is less resource-intensive, doesn't impact other elements on the page, and is disabled when it loses focus (perfect for reducing power usage!). You may think, therefore, that it makes sense to implement it in jQuery by default, right?

Exploring the requestAnimationFrame API's past

Ironically, jQuery used rAF back in version 1.6.2; it was pulled in 1.6.3, principally due to the animations piling up when windows regained focus. Part of this can be attributed to how rAF was (incorrectly) being used and that it would require major changes to its use, in order to correct the issues.

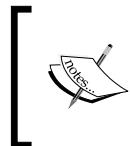


To see some of the issues with timing, browse to <http://xlo.co/requestanimationframe> – there are some demos on this site that illustrate perfectly why timing is so critical!



Using the requestAnimationFrame method today

Thankfully, we can still use requestAnimationFrame with jQuery today; Corey Frang, one of the developers of JQuery, wrote a plugin that can hook into and override the `setInterval()` method within the core library.



The original version of the plugin is available from GitHub for download at <https://github.com/gnarf/jquery-requestAnimationFrame/blob/master/src/jquery.requestAnimationFrame.js>.



This is probably one of the simplest changes we can make when using jQuery – we will explore this, and more, at the end of the exercise. For now, let's get on and create some code!

Creating our demo

For our next demo, we're going to use an updated version of a CodePen example, created by developer Matt West – the original demo is available from <http://codepen.io/matt-west/pen/bGdEC/>; I've refreshed the look and removed the vendor prefix elements of Corey's plugin, as they are no longer needed.

To give you some idea of what we're going to achieve, we will override the main `setInterval` method; this method does not call a jQuery method although it may look like it is; `setInterval` is a plain JavaScript function, as shown here:

```
jQuery.fx.start = function() {
    if ( !timerId ) {
        timerId = setInterval( jQuery.fx.tick, jQuery.fx.interval );
    }
};
```

I've incorporated a change of font too—for this demo, I've used the Noto Sans typeface, which can be downloaded from <http://www.fontsquirrel.com/fonts/noto-sans>; feel free to alter the code accordingly, if you want to use a different font.

Ready? Let's make a start by performing the following steps:

1. From a copy of the code download link that accompanies this book, go ahead and extract the `raf.css`, `raf.js` and `raf.html` files and save them to your project folder.
2. In a new file, add the following code—this is our monkey patch or a modified version of Corey's original plugin. We start by initiating a number of variables, as shown here:

```
(function( jQuery ) {
    var animating,
        requestAnimationFrame = window.requestAnimationFrame,
        cancelAnimationFrame = window.cancelAnimationFrame;

    requestAnimationFrame = window["RequestAnimationFrame"];
    cancelAnimationFrame = window["CancelAnimationFrame"];
```

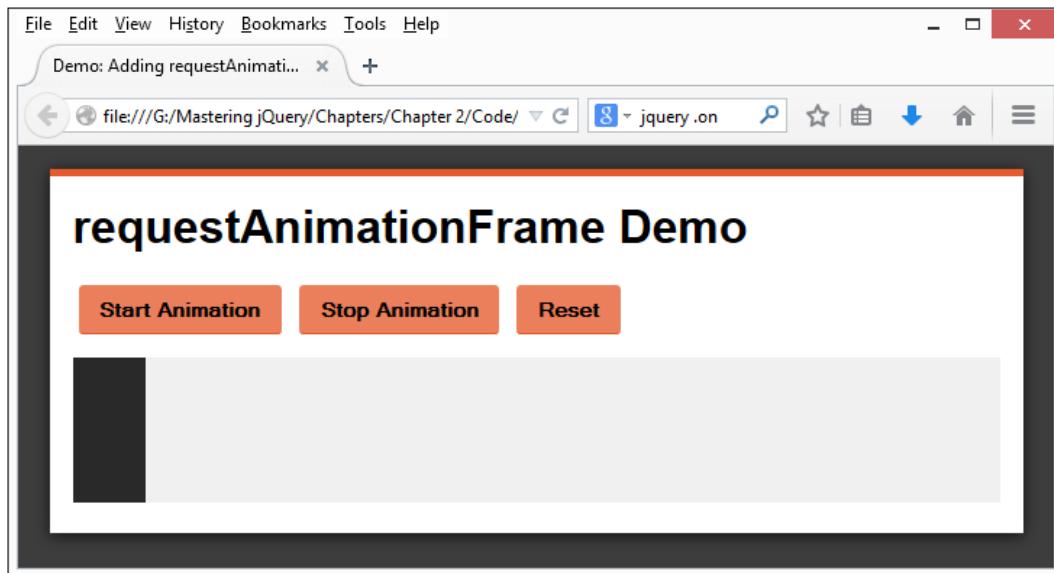
3. Next up comes the animating function, which is called from the main `requestAnimationFrame` method:

```
function raf() {
    if ( animating ) {
        requestAnimationFrame( raf );
        jQuery.fx.tick();
    }
}
```

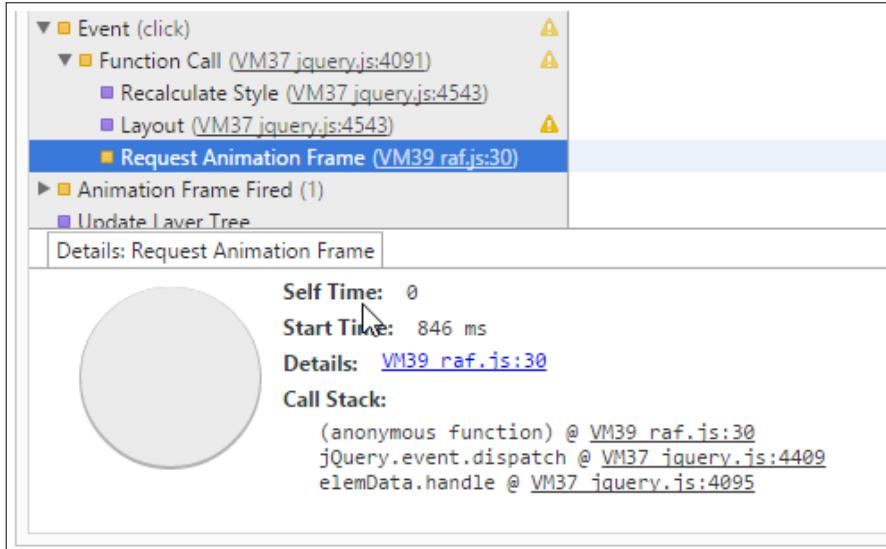
4. We now need our main `requestAnimationFrame` method; go ahead and add the following lines of code directly below the `raf()` event handler:

```
if ( requestAnimationFrame ) {  
  
    // use rAF  
    window.requestAnimationFrame = requestAnimationFrame;  
    window.cancelAnimationFrame = cancelAnimationFrame;  
    jQuery.fx.timer = function( timer ) {  
        if ( timer() && jQuery.timers.push( timer ) && !animating ) {  
            animating = true;  
            raf();  
        }  
    };  
    jQuery.fx.stop = function() {  
        animating = false;  
    };  
} ( jQuery );
```

5. Save the file as `jquery.requestAnimationFrame.js`, within a subfolder called `js` under the main project folder.
6. If you run the demo in a browser, you can expect to see the bar move when you press **Start Animation**, as shown in the following screenshot:



- To prove that the plugin is being used, we can use Google Chrome's **Timeline** option (within **Developer Tools**)—clicking on the red **Record** icon, then running the demo, and then stopping it produces this extract:



Make sure that you have the **JS Profiler** checkbox ticked under **Timeline** — the details will be shown; you may have to scroll down to view the **Event** entry.

This is probably one of the easiest changes we can make to override functionality in jQuery, yet potentially one of the most controversial—the latter being down to how we use it. The key point though is that we can override functionality using several formats.

The safest way is by the use of a plugin; in our example here, we used a modified one—the original was introduced from jQuery 1.8, so the changes made here just brought it up to the modern day. We could completely go in the opposite direction though and create a function that overrides an existing function—it carries a higher risk, but if done carefully, it is worth the effort! Let's take a look at a simple example, in the form of overriding `.hasClass()` to switch in the WebP format images when appropriate.

Adding WebP support to jQuery

At this point, I have a slight confession to make: adding full-blown WebP support to jQuery will probably be outside the scope of this book, let alone fill most of its pages!



WebP is a relatively new image format created by Google, which offers better compression than standard PNG files—you can read more about it at <https://developers.google.com/speed/webp/>. At present, both Chrome and Opera support this format natively; other browsers will display WebP images once support is added.

The next demo is really about how we can make the switch between two different ways of presenting content on screen, depending on whether our browser supports the newer format. A good example of this is where we might use CSS3 animation wherever possible and fall back to using jQuery for those browsers that do not support CSS3 animation natively.

In our next demo, we're going to use a similar principle to create a monkey patch that overrides the `.hasClass()` method in order to automatically switch to the WebP format images, where supported.



If you want to learn more, there is a useful discussion on how to get started with the format at <http://blog.teamtreehouse.com/getting-started-webp-image-format>.

Getting started

For the purpose of this demo, we will need to avail ourselves of an image in two different formats; I will assume JPEG has been used as our base format. The other image, of course, needs to be in the WebP format!

If you do not have the means already in place to convert your image to the WebP format, then you can do this using tools provided by Google, which are available for download at <https://developers.google.com/speed/webp/download>. Versions for Windows, Linux, and Mac OS are available for download here—for this exercise, I will assume that you are using Windows:

1. On the download page, click on <http://downloads.webmproject.org/releases/webp/index.html> and then look for `libwebp-0.4.2-windows-x64.zip` (if you are still using a 32-bit platform for Windows, then please select the `x86` version).

2. Once downloaded, extract the `libwebp-0.4.2-windows-x64` folder to a safe folder within your project folder and then navigate to the `bin` folder within it.
3. Open up a second Explorer view, then navigate to where your image is stored, and copy it into the `bin` folder.
4. Open up command prompt and then navigate to `C:\libwebp-0.4.2-windows-x64\bin`.
5. At the prompt, enter this command, replacing both the names with the names of your JPEG and WebP images, respectively:
`cwebp <name of JPG image> -o <name of WebP image>`
6. If all is well, we will get a screen similar to the following screenshot, along with our WebP format image in the `bin` folder:

```

C:\>cd libwebp-0.4.2-windows-x64
C:\libwebp-0.4.2-windows-x64>cd bin
C:\libwebp-0.4.2-windows-x64\bin>cwebp phalaenopsis.jpg -o phalaenopsis.webp
Saving file 'phalaenopsis.webp'
File: phalaenopsis.jpg
Dimension: 4050 x 2700
Output: 149858 bytes Y-U-U-All-PSNR 44.05 46.93 45.99 44.71 dB
block count: intra4: 17604
              intra16: 25322  (-> 58.99%)
              skipped block: 10976 (25.57%)
bytes used: header: 244 (0.2%)
             mode-partition: 48832 (32.6%)
Residuals bytes 1:segment 1:segment 2:segment 3:segment 4: total
  macroblocks: |   1%|   4%|  16%|  78%| 42926
    quantizer: |   36|   36|   30|   24|
    filter level: |   15|   23|   26|   21|
C:\libwebp-0.4.2-windows-x64\bin>

```

7. The last step is to copy the images into our project folder, so they are ready to be used for the next stage of our demo.

Creating our patch

Now that we have our images prepared, we can go ahead and set up the markup for our demo:

1. Go ahead and copy the following code into a new file, saving it as `replacewebp.html`:
- ```

<!DOCTYPE html>
<head>
 <title>Demo: supporting WebP images</title>

```

```
<script src="js/jquery.js"></script>
<script src="js/jquery.replacewebp.js"></script>
</head>
<body>

</body>
</html>
```

2. Next, we need to add in our monkey patch—in a new file, add the following code and save it as `jquery.replacewebp.js`. This is a little more involved, so we'll go through it in chunks, beginning with the standard declarations:

```
(function($){
 var hasClass = $.fn.hasClass;
 $.fn.hasClass = function(value) {
 var orig = hasClass.apply(this, arguments);
 var supported, callback;
```

3. Next comes the function that performs the test to see whether our browser supports the use of the WebP image format; add the following code immediately below the variable assignments:

```
function testWebP(callback) {
 var webP = new Image();
 webP.src = "data:image/webp;base64,UklGRi4AAABX"
+ "RUJQVlA4TCEAAAAvAUAAEB8wAiMw"
+ "AgSSNtse/cXjxyCCmrYNWPwmHRH9jwMA";
 webP.onload = webP.onerror = function () {
 callback(webP.height == 2);
 };
};
```

4. Next, we make use of the `testWebP` function to determine whether our browser can support the WebP image format—if it can, we alter the file extension used to `.webp`, as follows:

```
window.onload = function() {
 testWebP(function(supported) {
 console.log("WebP 0.2.0 " + (supported ? "supported!" : "not
supported."));
 $('.webp').each(function() {
 if (supported) {
 src = $(this).attr('src');
 $(this).attr('src', src.substr(0, src.length-3) + 'webp');
 console.log("Image switched to WebP format");
 }
 });
 });
};
```

```
 })
);
}
```

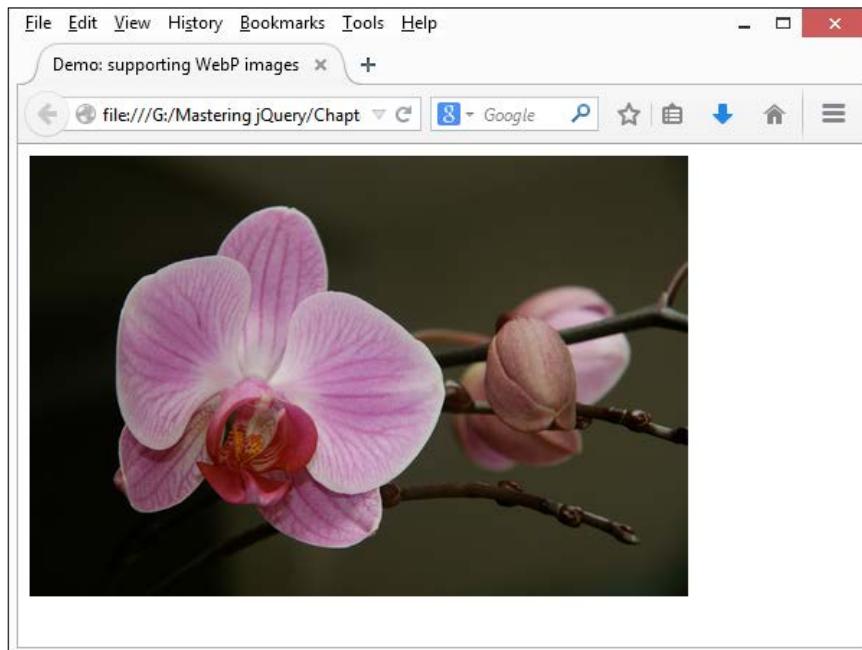
5. We finish off our function by executing the original version of our function, before terminating it with the closing brackets normally associated with an IIFE:

```
 return orig;
}
}) (:jQuery) ;
```

6. We then need to add one more function—this is used to initiate the call to `.hasClass()`; go ahead and add the following lines of code below the monkey patch function:

```
$(document).ready(function() {
 if ($("#img").hasClass("webp")) {
 $("#img").css("width", "80%");
 }
});
```

7. If all went well, when we run our demo, we will see an image of an Phalaenopsis, or Moth Orchid, as shown in the following screenshot:

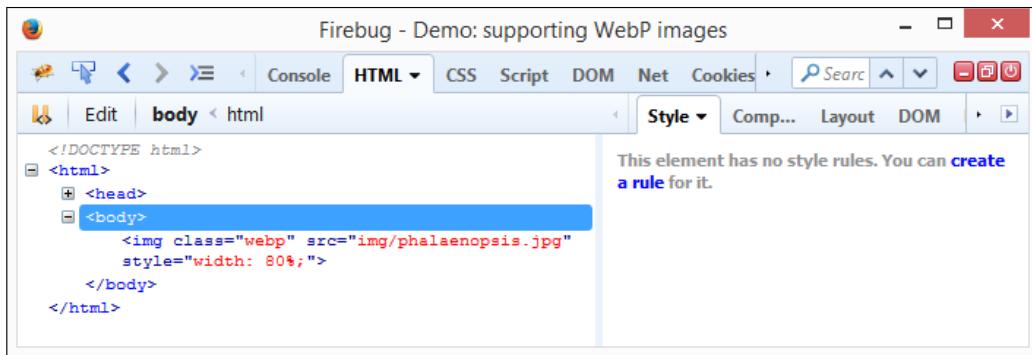


## *Customizing jQuery*

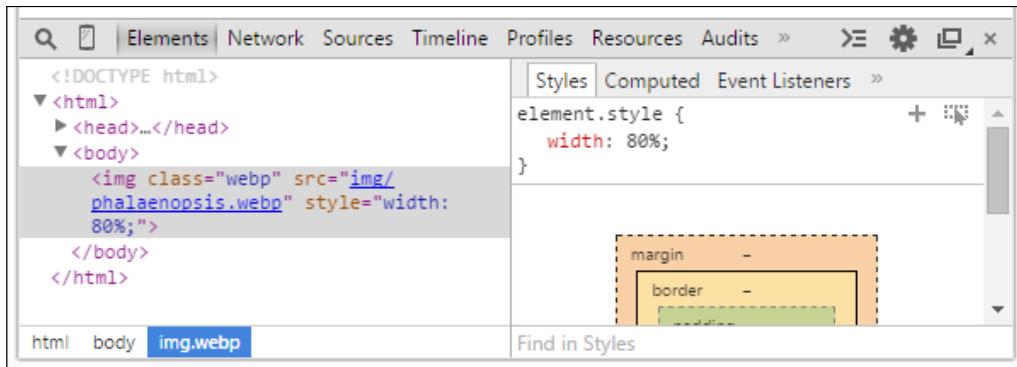
---

There is nothing out of the ordinary at this point; in fact, you're probably wondering what we've produced, right?

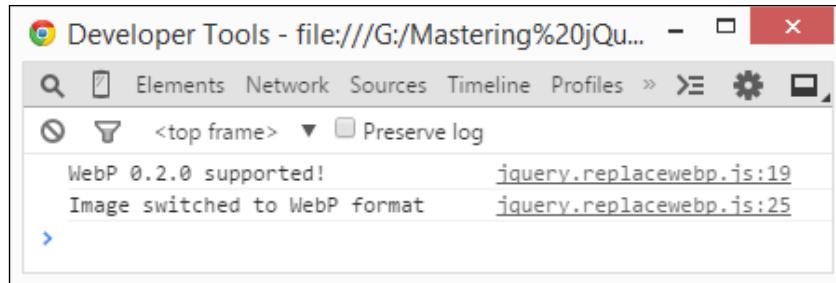
Aha! You'll see the answer to this question if you inspect the source using a DOM inspector, such as Firebug, as shown here:



Notice how it is showing a JPEG format image? That's because Firefox doesn't natively support this format out of the box; only Google Chrome does:



If you switch to using Google Chrome, then you can view the source by pressing *Ctrl + Shift + I*. You can clearly see the change in the format being used. If you are still in doubt, you can even take a look at the **Console** tab of Google Chrome. Here, it clearly shows that the patch has been referenced, as it displays the two messages that you expect to see:



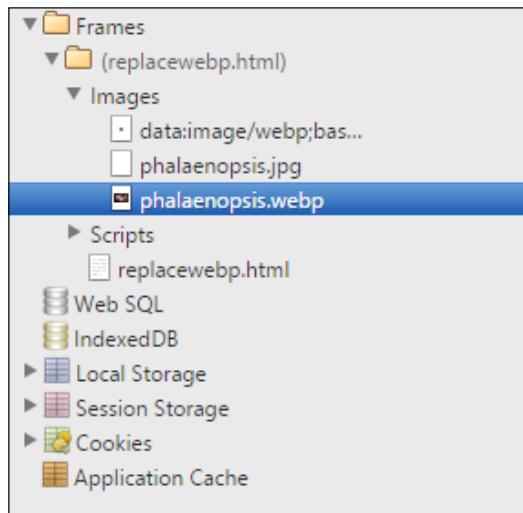
We've created our patch and it seems to work fine—that's all that we need to do, right? Wrong, there are more steps that we should consider, some of which may even prevent us from releasing the patch to a wider audience, at least for the time being.

There are some points that we need to consider and some actions that we may need to take as a result; let's pause for a moment and consider where we need to go from here, in terms of development.

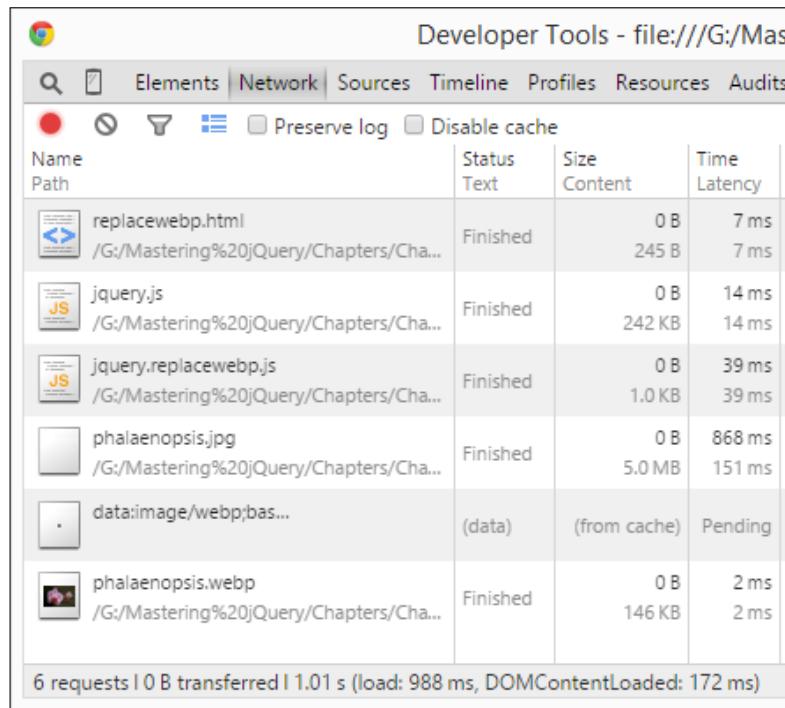
## Taking things further

In this example, we've overwritten an existing method as a means to illustrate duck punching—in reality, we will need to spend a little more time finessing our patch before we can release it!

The principle reason for this is the age-old issue of downloading more content than we really need; to prove this, take a look at the **Resources** tab of Google Chrome, when running the demo in that browser:



As if we need further confirmation, this extract from the **Timeline** tab also confirms the presence of both the JPEG and WebP images being called and the resulting impact on download times:



The screenshot shows the Network tab in the Chrome Developer Tools Timeline panel. It lists six requests with the following details:

Name	Status	Size	Time
Path	Text	Content	Latency
replacewebp.html /G/Mastering%20jQuery/Chapters/Cha...	Finished	0 B 245 B	7 ms 7 ms
jquery.js /G/Mastering%20jQuery/Chapters/Cha...	Finished	0 B 242 KB	14 ms 14 ms
jquery.replacewebp.js /G/Mastering%20jQuery/Chapters/Cha...	Finished	0 B 1.0 KB	39 ms 39 ms
phalaenopsis.jpg /G/Mastering%20jQuery/Chapters/Cha...	Finished	0 B 5.0 MB	868 ms 151 ms
data:image/webp;bas...	(data)	(from cache)	Pending
phalaenopsis.webp /G/Mastering%20jQuery/Chapters/Cha...	Finished	0 B 146 KB	2 ms 2 ms

6 requests | 0 B transferred | 1.01 s (load: 988 ms, DOMContentLoaded: 172 ms)

We created a patch here to illustrate what *can* be done; in reality, we will very likely include code to perform different actions on our content. As a start, we can do the following:

- Include support for more image formats – this can include JPEG, GIF, or SVG.
- Hardcode the code to accept one image format; we can extend the usability of our patch by making it more generic.
- jQuery is moving more toward a plugin-based architecture; should we really be considering patching the core code? There may be more mileage in creating a hook within the code, which then allows you to extend the existing functionality with a new plugin.
- We used `.hasClass()` as the basis for overriding an existing method; is this really the most appropriate thing to do? Although at face value it may appear to be useful, in reality, others may not agree with our choice of overriding `.hasClass` and consider other methods more useful.

There are plenty of questions that may be raised and need answering; it's only as a result of careful consideration that will we maximize any opportunity of our patch being successful and potentially consider it for submission to the core.

Let's change tack and switch to examining a key part of monkey patching. The process has its risks, so let's take a moment to consider some of these risks and the impact these may have on our work.

## Considering the pitfalls of monkey patching

Now that we've seen some examples in action, it's worth taking a moment to consider some of the risks of monkey patching libraries, such as jQuery:

- The principle risk and one that is likely to cause the most trouble is clashing. Imagine that you've created a patch that contains some functions – we'll call these functions 1, 2, and 3. Add another patch, and it is essential that we do *not* use the same function names; otherwise, it's difficult to determine whether function 1, or 2, or even 3 comes first?
- Another risk is security. If a library such as jQuery can be monkey patched, what is to stop anyone from introducing malicious constructs that damage the existing code? One can argue that this risk is always present in client-side scripting; the risk is greater when you override core jQuery functionality, compared to a standard plugin.
- There is always a risk that an upgrade to the core library may introduce a change that not only breaks your patch but also removes or alters the functionality that would otherwise provide a basis for your patch to work. This will prevent a site that uses jQuery from being upgraded and ultimately leave it vulnerable to attack.
- Adding too many patches without due care and consideration will make your API bloated and slow; this will reduce responsiveness and make it harder to manage, as we have to spend more time dissecting the code before we can get to the crux of an issue.
- Any monkey patches should really be kept within your site; they will be based on the code that directly alters jQuery, rather than use the predefined mechanism that a standard jQuery plugin provides. It is likely that authors may not have tested their monkey patches as extensively as they might have otherwise done for plugins; this presents a greater degree of risk if you are using someone else's patch.

- If a patch contains a large number of functions, then the impact of changing the core functionality is higher and wider; making these changes may break someone else's patch or plugin.

Ouch! There are serious concerns here! If we face these risks, then why use the process at all?

This is a good question; when used judiciously, monkey patching is a useful technique to help provide that little extra functionality or correct an issue. It can even act as a means of stress testing the code before it is submitted for committal. There's also an argument that says that the functionality should be included in a plugin, with good reason:

- Plugins can be released for use by others; they can contribute fixes or updates via sites such as GitHub, if the plugin is available
- Plugins are likely to work with a wider range of versions of jQuery than a simple patch; the latter is likely to be tailored to fix a specific issue
- Producing a patch to cover multiple fixes may result in a large file size or a lot of changes to the core functionality; this is better managed within the framework of a plugin, which can include other functionality, such as internationalization
- The jQuery Core is moving toward a leaner, faster architecture; adding lots of patches will increase the level of redundant functionality and make it less appealing for use to other developers

The key with monkey patching is not to abuse it; it is a valid tool but really only effective if you've exhausted all other possible solutions. If you have an urgent need for fixing an issue and cannot wait for an official update, then consider monkey patching jQuery—just be careful about how you do it!

## Distributing or applying patches

Once our patch is completed, we need to distribute it; it is tempting to simply update a version of jQuery and release that with our plugin or use it within our site. There are some disadvantages of using this method though:

- We can't take advantage of our browser's caching capabilities; if we use a cached version of jQuery, then it will either not contain our patched code or pull a fresh copy from the server.
- Patching a copy of jQuery means that we're locked into that version of jQuery. This prevents the end user from being able to use their own version of jQuery, a CDN link, or even a newer version of jQuery (assuming that the patch still works!).

- Allowing a patch to run separately at runtime means that it only patches the objects in the source code; if it goes horribly wrong, then we can drop the patch and still leave ourselves with a clean (unpatched) version of jQuery. Making changes to the source code does not afford us this luxury.

Instead, there are some alternatives that we can use to apply patches:

- We can simply include our patch in a separate file within our plugin or website – this keeps the Core jQuery library clean, although it means a slight overhead of requesting the patch file from the server. Users can then simply link to a copy of the file from runtime and discard if it circumstances change.
- Patches can also be distributed as a Gist – this makes it independent of our site or plugin and allows others to comment or suggest tweaks that can be incorporated into our code.

 As an example, I've created the following Gist for the `replacewebp.js` patch – this is available at <https://gist.github.com/alibby251/89765d464e03ed6e0bc1> and can be linked into projects as a means of distributing the code:

```
<script src="https://gist.github.com/alibby251/89765d464e03ed6e0bc1.js"></script>
```

- We can take this a step further if the patch is available within a GitHub repository – either as part of an existing project or on its own. GitHub will allow users to submit pull requests in order to help improve an existing patch before it is considered for submission to core.
- There is an alternative route that we can take: the patch can be packaged and delivered via a frontend package manager, such as Bower (<http://www.bower.io>) or Jam (<http://www.jamjs.org>).

 For more information on packaging content for download via Bower, please refer to <http://bower.io/docs/creating-packages/>.

These are some of the options we can use to distribute our patches; using a selection of these means that we can make our patch available to the widest possible audience and hopefully benefit from their testing and feedback!

## Summary

We've covered a lot of content in the last few pages, some of which may make your head spin, so let's take a breather and consider what we have learned.

We kicked off with an introduction to the patching of libraries, such as jQuery, and the term duck punching (or monkey patching). We looked at how we can replace or modify the existing behavior of jQuery by using this method, before moving on to create a basic monkey patch and working through its application to code.

Next up came a look at some of the benefits we can gain by using monkey patches; we spoke about the risk involved and some pitfalls that we need to consider when creating and applying patches.

We then switched to working through a number of demos that explored some of the ways in which we can alter code temporarily, before finishing with a look at how we can get our patches out into use for production.

Developing any form of patch or plugin requires well-maintained code if were to be successful. In the next chapter, we'll see how we can improve our skills in this area, with a look at using design patterns to better organize our code.

# 3

## Organizing Your Code

To organize or not organize, that's the question...

In our journey so far, we've covered the various means of downloading jQuery and seen how we can override core functionality with custom code, but—to misquote that famous detective: how should we organize our code?

Okay, you might think I'm losing the plot here, but bear with me on this; mastering a language such as jQuery is not just about producing complex code but about producing code that is well structured, concise, and clear to read.

In this chapter, we're going back to the basics with an introduction to some of the design patterns that are available within jQuery. We'll see how some of the techniques discussed in this chapter will help improve your code formatting and make you a better coder. In this chapter, we'll cover the following topics:

- Introducing design patterns and why we should use them
- Dissecting the structure of a design pattern
- Exploring some examples of the different design patterns available and the benefits of using them
- Exploring the use of patterns within the jQuery library

Ready to get started? Let's begin...

## Introducing design patterns

How many times have you viewed a website to stand in awe at the beautiful design, only to find that the code looks like the proverbial dog's dinner? A commonly held misconception is that appearance is the domain of the designer; this is not true, as the design of the code plays an equally important part as well.

How can we get around this? Easy, we can use a **design pattern** or a set of constructs that help provide a solution and allow us to concentrate more on the functionality we want to provide within our project.

First created in 1977 by the architect Christopher Alexander, engineers have since used the early principles and developed them into what we now know as design patterns. This work was further promoted by the **Gang of Four (GoF)** in their iconic book *Design Patterns: Elements of Reusable Object-Oriented Software*, published in 1995.

They helped to not only push the use of design patterns further afield but also provided some design techniques and pitfalls; they were also instrumental in providing the twenty-three core patterns that are frequently used today (of which, we will be covering those that are used within jQuery development). We'll take a look at some of the patterns that are in use today, but first, let's answer a simple question. What do we really mean by a design pattern and how do they help us to write clear and concise code, which reduces unnecessary duplication?

## Defining design patterns

At a basic level, design patterns take the format of predefined templates or a set of reusable principles that help classify different approaches, as part of supporting good design.

Why use them? Simple, there are three main benefits to incorporating a design pattern into our project:

- **Design patterns are proven solutions:** They are based on solid approaches to solving an issue in software development and are based on the experience of developers who help create the pattern being used
- **Patterns can be reused:** Although they often represent an out-of-the-box solution, they can be easily adapted as per our needs
- **Patterns are expressive:** They contain a set structure and vocabulary to help you express large solutions clearly and elegantly

At this point, you may be forgiven for thinking that patterns must be an exact science, where we're constrained by the framework of the pattern we're using. It's not so; they are not an exact solution but merely a scheme to help provide a solution.

Taking it further, there are a number of other reasons why we should consider using design patterns in our work:

- We can effectively code out or prevent minor issues that might cause us major problems later in the development process – using tried and tested techniques eliminates the need to worry about the structure of our code and allows us to concentrate on the quality of our solution.
- Patterns are designed to provide generalized solutions that don't tie them to a specific problem but can be applied to improve the structure of our code.
- Some patterns, if chosen wisely, can help reduce the amount of code by avoiding repetition; they encourage us to look at our code carefully, to reduce duplication and keep to using **Don't Repeat Yourself (DRY)** principles that are one of the tenets of jQuery.
- Patterns are not a one-shot, point-in-time solution; our work may help improve existing designs or even provide scope for creating new patterns! This constant improvement helps ensure that patterns become more robust over time.

Irrespective of its purpose, one of the key tenets of design patterns is that they are not always considered a design pattern, unless they have been rigorously tested by the pattern community. Many might appear to be a pattern; in reality, they are more likely to be a proto-pattern or a pattern that has been created but not yet been sufficiently tested to be classed as a true pattern.

The core principle of any design pattern is based on Alexander's belief that they should always represent a process and an output. The latter term is deliberately meant to be vague; it should represent something visual but the exact context of what that visual output is will depend on the chosen pattern.

So, now that we've seen what a design pattern is, let's discuss what they look like. Are they made up of specific elements or constructs? Before we take a look at some examples, let's first consider the makeup of a design pattern and how we can use it to good effect.

## Dissecting the structure of a design pattern

If you take a look at any design pattern in detail, you will find that it is made up of a rule that establishes a relationship between the following:

- A context
- A system of forces that arises in that context
- A configuration that allows these forces to resolve themselves in the context

These three key aspects can be further broken down into a number of different elements, in addition to a pattern name and description:

Element	Purpose or function
Context outline	The context in which the pattern is effective in responding to the users' needs.
Problem statement	A statement of the problem being addressed, so we can understand the intent of the pattern.
Solution	A description of how the user's problem is being solved in a list of steps and perceptions that are easy to understand.
Design	A description of the pattern's design and, in particular, the user's behavior when interacting with it.
Implementation	A guide to how the pattern will be implemented.
Illustrations	A visual representation of the classes in the pattern, such as a UML diagram.
Examples	An implementation of the pattern in a minimal form.
Corequisites	What other patterns may be needed to support the use of the pattern being described?
Relations	Does this pattern resemble (or mimic) any existing ones?
Known usage	Is the pattern already being used in the wild? If so, where and how?
Discussions	The team or the author's thoughts on the benefits of using the pattern.

The beauty about using patterns is that while they may entail a degree of effort during the planning and documentation stages, they are useful tools that help to get all the developers in a team on the same page.

It is worth taking a look at the existing patterns first, before creating new ones – there may be one in use already, which reduces the need to design from scratch and go through a lengthy process of testing before being accepted by other developers.

## Categorizing patterns

Now that we've seen the structure of a typical design pattern, let's take a moment to consider the types of patterns that are available. Patterns are usually grouped into one of the following three categories, which are the most important ones:

- **Creational patterns:** These focus on how we can create objects or classes. Even though this might sound simple (and in some aspects, like common sense), they can be really effective in large applications that need to control the object creation process. Examples of creational patterns include Abstract, Singleton, or Builder.
- **Structural design patterns:** These focus on ways to manage relationships between objects so that your application is architected in a scalable way. A key aspect of structural patterns is to ensure that a change in one part of your application does not affect all the other parts. This group covers patterns such as Proxy, Adapter, or Façade.
- **Behavioral patterns:** These focus on communication between objects and include the Observer, Iterator, and Strategy patterns.

With this in mind, let's take a moment to explore some of the more commonly used designs, beginning with the **Composite Pattern**.

## The Composite Pattern

If you've spent time developing with jQuery, how often have you written code similar to this:

```
// Single elements
$("#mnuFile").addClass("active");
$("#btnSubmit").addClass("active");

// Collections of elements
$("div").addClass("active");
```

Without realizing it, we're using two instances of the Composite Pattern—a member of the Structural group of patterns; it allows you to apply the same treatment to a single object or a group of objects in the same manner, irrespective of how many items we're targeting.

In a nutshell, when we apply methods to an element, or a group of elements, a jQuery object is applied; this means that we can treat either set in a uniform manner.

So, what does this mean? Let's take a look at a couple of other examples:

```
// defining event handlers
$("#tablelist tbody tr").on("click", function(event) {
 alert($(this).text());
});
$('#btnDelete').on("click", function(event) {
 alert("This item was deleted.");
});
```

The beauty of using Composite Patterns is that we can use the same method in each instance but apply different values to each element; it presents a uniform interface to the end user while applying the change seamlessly in the background.

## **Advantages and disadvantages of the Composite Pattern**

Using Composite Patterns can be as simple or complex as we make it; there are advantages and drawbacks to using this pattern, which we should consider:

- We can call a single function on a top-level object and have it apply the same results to any or all of the nodes within the structure
- All the objects in the composite design are loosely coupled, as they all follow the same interface
- The composite design gives a nice structure to the objects, without the need to keep them in an array or as separate variables

There are some drawbacks to using composite patterns; the following are the key ones to be considered:

- We can't always tell whether we're dealing with a single item or multiple items; the API uses the same pattern for both
- The speed and performance of your site will be affected, if the composite pattern grows beyond a certain size

Let's move on and take a look at some more patterns; the next one up is the **Adapter Pattern**.

## The Adapter Pattern

We can use jQuery to switch classes assigned to selectors; in some cases though, this may be an overkill for our needs, or assigning a class to a selector may present issues that we need to avoid. Fortunately, we can use the `.css()` function to directly apply styles to our elements—this is a great example of using an Adapter Pattern in jQuery.

A pattern based on the Structural design pattern, the Adapter Pattern, translates the interface for an element in jQuery into an interface that is compatible with a specific system. In this case, we can assign a CSS style to our chosen element, using an adapter in the form of `.css()`:

```
// Setting opacity
$(".container").css({ opacity: 0.7 });

// Getting opacity
var currentOpacity = $(".container").css('opacity');
```

The beauty of this is that once the style is set, we can use the same command to get the style value.

## Advantages and disadvantages of the Adapter Pattern

There are several key benefits of using the Adapter design pattern; the key one being its ability to link two incompatible interfaces, which would otherwise have had to remain independent.

In addition, it is worth making a note of the following additional benefits:

- The Adapter pattern can be used to create a shell around an existing block of code, such as a class, without affecting its core functionality
- This pattern helps makes the code reusable; we can adapt the shell to include additional functionality or modify the existing code if circumstances dictate a need to do so

Using an Adapter Pattern presents some drawbacks, if we're not careful:

- There is a performance cost in using keywords such as `.css()`—do we really need to use them? Or, can we apply a style class or selector and move CSS styling into the style sheet instead?

- Using keywords, such as `.css()`, to manipulate the DOM can lead to a performance hit if we have not simplified our selectors and if we've used something like this:

```
$(".container input#elem").css("color", "red");
```

This likely to not be noticeable on a small site or where such manipulation is only lightly used; it will be noticeable on a larger site!

- Adapter patterns allow you to chain jQuery commands; although this will help reduce the amount of code that needs to be written, it comes with a trade-off in terms of legibility. Chaining commands will make it harder to debug code at a later date, particularly if there is a change in the developer involved; there's something to be said for keeping code simple and clean, if only to help maintain one's sanity!

Let's move on and take another look at another pattern, namely the **Façade Pattern**.

## The Façade Pattern

Originally from French, façade translates as *frontage* or *face*—this is a perfect description for this next pattern; its outward appearance can be very deceptive, in just the amount of code that can be hidden!

The **Façade Pattern**, another member of the Structural group of patterns, provides a simple interface to a larger, more complex body of code; in a sense, it abstracts some of the complexity, leaving us with simple definitions that we can manipulate at will. Notable examples of Façade Patterns are DOM manipulation, animation, and, of course, the perennial favorite, AJAX!

As an example, simple AJAX methods such as `$.get` and `$.post` both call the same parameters:

```
$.get(url, data, callback, dataType);
$.post(url, data, callback, dataType);
```

These are façades to two more functions in their own right:

```
// $.get()
$.ajax({
 url: url,
 data: data,
 dataType: dataType
}).done(callback);
```

```
// $.post
$.ajax({
 type: "POST",
 url: url,
 data: data,
 dataType: dataType
}).done(callback);
```

Which in turn are façades to a huge amount of complex code! The complexity in this instance stems from the need to iron out cross-browser differences for XHR and make it a cinch to work with actions, such as `get`, `post`, `deferred`, and `promises` in jQuery.

## Creating a simple animation

At a very simple level, the `$.fn.animate` function is an example of a façade function in jQuery, as it uses multiple internal functions to achieve the desired result. So, here's a simple demo that uses animation code:

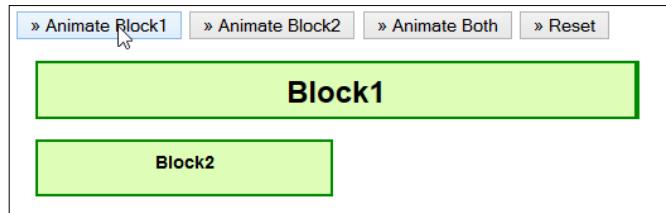
```
$(document).ready(function() {
 $("#go1").click(function() {
 $("#block1")
 .animate({width: "85%"}, {queue: false, duration: 3000})
 .animate({fontSize: "24px"}, 1500)
 .animate({borderRightWidth: "15px"}, 1500);
 });

 $("#go2").click(function() {
 $("#block2")
 .animate({ width: "85%" }, 1000)
 .animate({ fontSize: "24px" }, 1000)
 .animate({ borderLeftWidth: "15px" }, 1000);
 });

 $("#go3").click(function() {
 $("#go1").add("#go2").click();
 });

 $("#go4").click(function() {
 $("div").css({width: "", fontSize: "", borderWidth: ""});
 });
})
```

The preceding code will produce this animation effect:



We can make use of the function shown in the following screenshot within the core library:

```
6681 animate: function(prop, speed, easing, callback) {
6682 var empty = jQuery.isEmptyObject(prop),
6683 optall = jQuery.speed(speed, easing, callback),
6684 doAnimation = function() {
6685 // Operate on a copy of prop so per-property easing won't be lost
6686 var anim = Animation(this, jQuery.extend({}, prop), optall);
6687
6688 // Empty animations, or finishing resolves immediately
6689 if (empty || data_priv.get(this, "finish")) {
6690 anim.stop(true);
6691 }
6692 };
6693 doAnimation.finish = doAnimation;
6694
6695 return empty || optall.queue === false ?
6696 this.each(doAnimation) :
6697 this.queue(optall.queue, doAnimation);
6698 },
6699 },
```



The code for the demo in this section is available in the code download link that accompanies this book, as the `animation.html` file; you will need to extract the whole code folder for this demo to work correctly.

Now that you've seen the Façade Pattern in use, let's consider some of the benefits of using it in our code.

## Advantages and disadvantages of the Façade Pattern

Using the Façade pattern to hide complex code is a really useful technique; in addition to being easy to implement, there are other advantages of using this pattern, as follows:

- Enhances security of your web application
- Works well in combination with other patterns

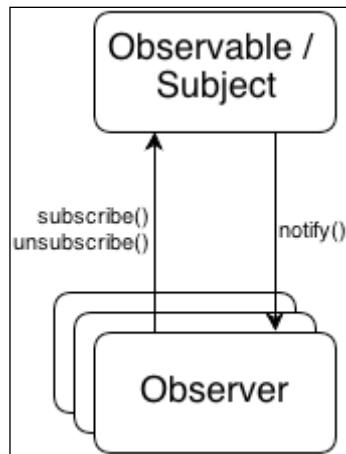
- Makes it easy to patch internal code
- Provides a simpler public interface to the end user

In contrast to other patterns, there are no real notable drawbacks when using this pattern; it provides a unified set of interfaces to us as end users, so we're not forced to make any compromises. It is worth noting that there is a cost involved in implementation, when abstracting code – this is something that we should always bear in mind when using façade patterns.

## The Observer Pattern

Since it is a member of the Behavioral group of patterns, we will already be familiar with this next pattern – if you spend time creating custom events, then you are already using the **Observer Pattern**.

A key part of developing with jQuery is using its well-established publishing/subscribing system to create custom events – access to these events is possible using `.trigger()`, `.on()`, or `.off()`. We can define Observer Patterns as those patterns where specific objects are subscribed to others and can be notified by them when a particular event takes place:



For a moment, let's say we have the following HTML:

```
<div id="div1">This is div 1</div>
<div id="div2">This is div 2</div>
```

We want the inner `<div>` elements to trigger an event called `customEvent`; this will happen when they are clicked on:

```
$('div').on('click', function(e) {
 $(this).trigger('customEvent');
});
```

Now, let's make the document element subscribe to `customEvent`:

```
$(document).on('custom', function(e) {
 console.log('document is handling custom event triggered by ' +
 e.target.id);
});
```

When the custom event is triggered by one of the `div` elements, the observer/subscriber is notified and a message is logged to the console.



For purists, some of you may prefer to use a typical publish/subscribe model – an example is available at <https://gist.github.com/cowboy/661855>.



Let's consider some of the benefits of using this pattern and where you may need to make allowances in your code in order to avoid falling into some of the traps associated with using this design pattern.

## Advantages and disadvantages of the Observer Pattern

Using the Observer Pattern forces us to consider the relationship between the various components of an application, at a far greater level than we might otherwise have been used to considering. It is also great at doing the following:

- Promoting loose coupling in jQuery, where each component knows what it is responsible for and doesn't care about other modules – this encourages reusable code.
- Allowing you to follow the separation of concerns principle; if code blocks are self-contained, they can be reused with little difficulty in new projects. We can then subscribe to single events and not worry about what happens in each block.
- Helping us to pinpoint where the dependencies are within our project, as a potential basis for determining whether these dependencies can be reduced or eliminated altogether with a little effort.

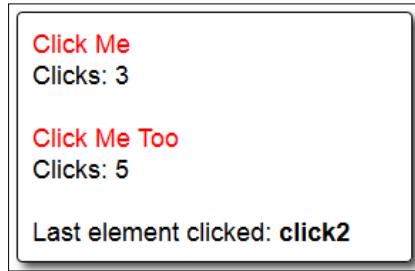
There are drawbacks to using the Observer Pattern though; the key drawbacks are the switching of a subscriber from one publisher to another can be costly in terms of code, and it becomes harder to maintain the integrity of our code.

To illustrate this, let's take a brief look at a simple example, where we can see at least one instance of where we've had to make extra allowances for the switch of publisher.

## **Creating a basic example**

Getting our heads around how the Observer Pattern works is critical; it is one of the patterns that is more in-depth and provides more opportunity than a simple set of protocols, such as the Façade design pattern. With this mind, let's run through a quick working demo to illustrate how it works, as shown here:

- Let's start by downloading and extracting copies of the code for this chapter—we need the `observer.html` file, along with the `css` and `js` folders
- If you run the demo, you should see two labels in red, which you can click; if you try clicking them, you will see the counts increase, as shown in this screenshot:



At this point, let's consider the code—the key functionality is in the `observer.js` file, which I have reproduced in full here:

```
$ (document) . ready (function () {
 var clickCallbacks = $.Callbacks ();
 clickCallbacks.add(function () {
 var count = parseInt (this.text (), 10);
 this.text (count + 1);
 });
 clickCallbacks.add(function (id) {
 $('span', '#last').text (id);
 });
 $('.click').click(function () {
```

```
var $element = $(this).next('div').find('[id^="clickCount"]');
clickCallbacks.fireWith($element, [this.id]);
});
});
```

Notice how there is a single event handler for the `.click` class. We've used a callback here to allow jQuery to execute the next click, even though it may not have finished completing the previous execution. In this instance, it's not going to be too much of an issue, but if we had to update a number of different statements or apply more changes (through the use of additional functions), then the callback will prevent errors from being generated in our code.

Here, we subscribe to the observable, which—in this instance—are the two **Click Me** statements; the `.click` event handler allows us to update both the click counts and the **Last element clicked** statement, without throwing an error.



To learn more about the intricacies of using callbacks in jQuery, you may want to browse through the API documentation, which can be viewed at <http://api.jquery.com/jquery.callbacks/>.

In the meantime, let's change focus and take a look at a different pattern. We all know that jQuery is famed for its DOM manipulation abilities; up next is the Iterator pattern, which is based on this particular feature of jQuery.

## The Iterator Pattern

Now, how many times have you heard, or read, that jQuery is famed for its DOM manipulation? I bet that it's a fair few times and that the `.each()` keyword is used at some point in those examples.

DOM manipulation in jQuery uses a special variation of the Iterator Pattern, from the Behavioral group of patterns—this is as it sounds; we can use this pattern to traverse (or iterate) through all the elements of a collection, leaving jQuery to handle the internal workings. A simple example of such a pattern might look like this:

```
$.each(["richard", "kieran", "dave", "alex"], function (index, value) {
 console.log(index + ": " + value);
});

$("li a").each(function (index) {
 console.log(index + ": " + $(this).text());
});
```

In both the cases, we've used the `.each` function to iterate through either the array or each instance of the `li` selector; there is no need to worry about the internal workings of the iterator.

Our examples contain minimal code, in order to iterate through each selector or class within the page; it's worth taking a look at the amount of code that is behind `jQuery.fn.each()` function, within the core library:

```
138 each: function(callback, args) {
139 return jQuery.each(this, callback, args);
140 },
```

This, in turn, calls the `jQuery.each()` function—the first one is for internal use only, as shown in the following screenshot:

```
346 each: function(obj, callback, args) {
347 var value,
348 i = 0,
349 length = obj.length,
350 isArray = isArraylike(obj);
351
352 if (args) {
353 if (isArray) {
354 for (; i < length; i++) {
355 value = callback.apply(obj[i], args);
356
357 if (value === false) {
358 break;
359 }
360 }
361 } else {
362 for (i in obj) {
363 value = callback.apply(obj[i], args);
364
365 if (value === false) {
366 break;
367 }
368 }
369 }
370 }
371 },
```

This is then supplemented by a special fast case, that is, for the most common use of the `.each()` function:

```
371 // A special, fast, case for the most common use of each
372 } else {
373 if (isArray) {
374 for (; i < length; i++) {
375 value = callback.call(obj[i], i, obj[i]);
376
377 if (value === false) {
378 break;
379 }
380 }
381 } else {
382 for (i in obj) {
383 value = callback.call(obj[i], i, obj[i]);
384
385 if (value === false) {
386 break;
387 }
388 }
389 }
390 }
391
392 return obj;
393 },
```

## Advantages and disadvantages of the Iterator Pattern

The ability to iterate over elements in the DOM is one of the key elements of jQuery—as a critical part of the Iterator pattern; these are some benefits of using this pattern:

- The Iterator pattern hides much of the functionality required to iterate through a collection, without the need to understand the inner workings of the code providing this functionality
- We can use the same consistent pattern to iterate through any object or set of values
- Using the Iterator process can also help to reduce or eliminate typical `for` loop syntax across our code, making the code easier to read

Unlike other patterns, there are very few disadvantages of using this pattern. It's a key facet of jQuery, so provided it is not abused by having to iterate over an excessive number of objects, this simple pattern will prove to be very useful!

## The Lazy Initialization Pattern

Hehe, this sounds like something I might follow on a Sunday morning! Okay, I know that was a terrible joke, but all jokes aside, this creational-based pattern allows you to postpone expensive processes until they are needed.

At its simplest level, we might configure a plugin with a number of different options, such as the number of images to be displayed, whether we should show an overlay, or how each image is displayed. Sounds simple, right? So, where does the lazy initialization part come in? Aha! This is simpler than you might think. Take the example of the following code:

```
$ (document) .ready(function() {
 $("#wowslider-container1") .wowSlider();
});
```

Our example used the initialization command for the WOW Slider (available from <http://www.wowslider.com>)—the key to using this pattern is in the initialization process; it is not fired until the first moment it is needed on our page.

A more complex example of the lazy initialization pattern is a callback; these won't be processed until the DOM is ready:

```
$ (document) .ready(function () {
 var jqxhr = $.ajax({
 url: "http://domain.com/api/",
 data: "display=latest&order=ascending"
 })
 .done(function(data)){
 $(".status") .html("content loaded");
 console.log("Data output:" + data);
 });
});
```

We might make use of this example directly in our code; it is more likely that we will use it within a lazy loading plugin, such as the version by Mika Tuupola at <http://www.appelsiini.net/projects/lazyload>.

## Advantages and disadvantages of the Lazy Initialization Pattern

The key benefit of using this design pattern is simple: to delay the loading of expensive resources until they are needed; this helps to speed up access to a site and to reduce bandwidth usage (at least initially).

However, there are some drawbacks of using this method, which include the following:

- It needs careful management by the setting of a flag to test whether the summoned object is ready for use; if not, then a race condition can be generated in multithreaded code
- The prior use of any lazy variable or object will bypass the initialization on the first access principle and mean that we lose any benefit of not loading these large objects or variables
- This method requires the use of a map to store instances so that you get the same instance when you next ask for one with the same parameter as the one previously used
- There is a time penalty involved with using this pattern, if large objects need to be loaded; the pattern only really works if these objects are not loaded initially and if there is a good chance that they will not be used

Ultimately, using this pattern needs some consideration and careful planning; it will work well, provided we've chosen not to load the right objects, so to speak! Talking of strategy, let's move on and take a look at another pattern that helps us determine what will happen when changing states on objects or variables, namely the **Strategy Pattern**.

## The Strategy Pattern

Cast your mind back a few years, when using Flash to animate content on sites was the latest design fad; there were some really well-designed examples, although all too often sites were slow and not always as effective as they should be! Moving forward, CSS animations are preferred now – they don't need a browser plugin to operate, can be stored in a style sheet, and are less resource hungry than Flash.

"Why are we talking about animations?", I hear you ask, when this chapter is about design patterns. That's a good question; the answer is simple, though: some of you may not realize it but animations are a perfect example of our next design pattern. At a basic level, animations are all about changing from one state to another – this forms the basis of the Strategy pattern, from the Behavioral group of patterns.

Also known as the policy or state pattern, the Strategy pattern allows you to select the appropriate behavior at runtime. In a nutshell, this is what the pattern does:

- Defines a family of algorithms (or functions) used to determine what should happen at runtime
- Encapsulates each algorithm (or function) into its self-contained unit and makes each algorithm interchangeable within that family

A good example of where strategy patterns can be used is in the validation of entries in a form—we need some rules in place to determine what is valid or invalid content; we clearly won't know what the outcome will be until that content is entered!

The key point here is that the rules for validation can be encapsulated in their own block (potentially as an object in their own right); we can then pull in the relevant block (or rule), once we know what the user wants us to validate.

At a more basic level, though, there is a simpler example of a strategy pattern; it takes the form of animating content, such as using `.toggle()`, where we switch from one state to another or back again:

```
$('div').toggle(function() {}, function() {});
```

Each resulting state can be set as a class in its own right; they will be called at the appropriate time, once we know what the requested action should be. To help set the context, let's knock up a simple demo in order to see this in action.

## Building a simple toggle effect

Okay, granted that this is jQuery 101, but why complicate matters when it shows what we need perfectly?

In this demo, we perform a simple toggle action to show or hide two `<p>` statements—the key point here is that we don't know what is going to happen next, until the button is pressed.

To see this in action, download a copy of the code folder for this chapter; run the `strategy.html` demo and then click on **Toggle 'em** to see the `<p>` statements appear or disappear, as shown here:



The magic takes place in this function; it's a simple use of the `.toggle()` command to switch the visibility of each `<p>` statement as required:

```
$(document).ready(function() {
 $("button").click(function() {
 $("p").toggle("slow");
 });
});
```

However, we can easily abstract the function contained in the click event handler into a separate IIFE and then simply call the function in our code, as shown here:

```
$(document).ready(function() {
 var hideParagraphs = function() {
 $("p").toggle("slow");
 };

 $("button").click(hideParagraphs);
});
```

The code is already easier to read—we've removed the bulk of the original action away from the event handler; this removes the need to edit the event handler if we need to change the code at a later date.



If you are interested in learning more about IIFEs, then you may want to take a look at Wikipedia's entry for more details, which is available at [https://en.wikipedia.org/wiki/Immediately-invoked\\_function\\_expression](https://en.wikipedia.org/wiki/Immediately-invoked_function_expression).

## Switching between actions

Although we've concentrated on animations in our example, the observant amongst us might be wondering whether the same techniques will apply to commands such as `switch()`. The answer is yes; we've not discussed it here as it is a pure JavaScript command, but you can apply the same principles as an alternative to use it.

## Advantages and disadvantages of the Strategy Pattern

Defining a sensible strategy is the key to successful coding; these are some benefits that we can gain by using the Strategy pattern:

- The code is easier to read; if we abstract functions into their own class, we can move them away from the decision-making process, either as separate blocks of code in the same file or even as separate files in their own right

- The code is easier to maintain; we only need to go to the class to change or refactor the code, and we only need to make minimal changes to the core code in order to add links to new classes or object event handlers
- We can maintain the separation of concerns—each independent class or object that we abstract retains no awareness of the other components, but when provided with each strategy object's responsibility and the same interface, they can communicate with other objects
- Using the Strategy pattern allows you to take advantage of the open/closed principle; the behavior of each abstracted class or object can be altered by initiating a new class or object instance of the existing behaviors

[  For more details about the open/closed principle, please refer to [http://en.wikipedia.org/wiki/Open/closed\\_principle](http://en.wikipedia.org/wiki/Open/closed_principle). ]

These are some of the disadvantages that we need to be mindful of, though:

- Use of the Strategy pattern allows you to respect the open/closed principle, but at the same time, you may end up initiating a new class or object of code that contains a lot of unnecessary functions or actions that make your code more cumbersome
- There may be instances where using the Strategy pattern won't suit your purposes; if your code only contains a small number of functions, the effort required to abstract them may outweigh the benefits of doing so
- Using the Strategy pattern will increase the number of objects within our code, making it more complex and potentially requiring more resources to manage

Enough strategizing for the moment; let's move on and take a look at a different protocol, in the form of the **Proxy** design pattern.

## The Proxy Pattern

When working with jQuery, there will be occasions where you might want to write a generic event handler that takes care of managing styles on certain elements—a good example might be switching from active to disabled state, or even selected state; we can then style these using normal CSS.

Using this approach, this is how a generic event handler might look:

```
$(".myCheckbox").on("click", function () {
 // Within this function, "this" refers to the clicked element
 $(this).addClass("active");
});
```

At face value, this will work perfectly well, but what if we were to introduce a delay before changing the style class? We would normally use a `setTimeout()` function to achieve this:

```
$(".myCheckbox").on("click", function () {
 setTimeout(function () {
 // "this" doesn't refer to our element, but to the window!
 $(this).addClass("selected");
 });
});
```

Did anyone spot a small but rather crucial problem here? Passing any function to `setTimeout` will give you the wrong value—it will refer to the window object, not the object being passed!

A workaround for this is jQuery's `proxy()` function; we can use this function to implement a Proxy pattern, or a go-between, in order to ensure that the right value is passed through to the `.addClass()` method in the right context. We can adapt our previous example as shown in the following code snippet:

```
$(".myCheckbox").on("click", function () {
 setTimeout($.proxy(function () {
 // "this" now refers to our element as we wanted
 $(this).addClass("active");
 }, this), 500);
});
```

The last `this` parameter we're passing tells `$.proxy()` that our DOM element is the value we want `this` to refer to—in this instance, it's the checkbox and not the window.

## Advantages and disadvantages of the Proxy Pattern

Proxy patterns are useful designs from the Structural group, which can help with optimizing and maintaining fast sites; at its core, the pattern is based on the principle of not loading expensive elements until absolutely necessary. (and then ideally not loading at all, if it can be helped!)

There are some benefits we can gain by using this design pattern, as follows:

- We can use a proxy pattern to provide a placeholder for more expensive objects that are yet to be loaded or which may never be loaded; this includes objects that may be loaded from outside the application
- Using a proxy can act as a wrapper, providing delegation to the real object, while also protecting it from undue complexity
- Incorporating proxy patterns into our page can help reduce the perceived slowness or lack of responsive from code-heavy sites

The downsides to using this pattern include the following:

- There is a risk that a proxy pattern can hide the life cycle and state of a volatile resource from its client; this means that the code has to wait until the right resource becomes available again or produces an error. It needs to know that it is interacting with the original resource and not with another resource that may appear similar to the original.
- If we are using proxy patterns to represent a remote resource, this will disguise the use of communication between the two; communication with a remote resource should be treated differently to that of a local one.

With care, proxy patterns can prove very useful, provided we're sensible about what we decide to load or not load into our pages. Let's change tack and look at another design pattern; this one is based on how we may need to construct one or more elements dynamically; this concept is at the core of the **Builder Pattern**.

## Builder Pattern

During the development of any project, there may be occasions where we need to create new elements dynamically; this can range from building a single `<div>` element to a complex mix of elements.

We might want the flexibility of defining the final markup directly in our code, which can get messy, or we can separate out the elements into a standalone mechanism that allows us to simply build those elements, ready for use later in the code.

The latter, or the Builder Pattern to give it its technical name, is preferable; it's easier to read and allows you to keep a clear distinction between variables and the rest of your code. This particular pattern falls into the Creational group of patterns and is one of the few common examples you will see of this type of pattern.



You may see references to the **Abstract Pattern** online, or in books—it is very similar in style to the Builder Pattern.

We can use jQuery's dollar sign to build our objects; we can either pass the complete markup for an element, partial markup and content, or simply use jQuery for construction, as shown here:

```
$('<div class="foo">bar</div>');

$('<p id="newText">foo bar</p>').appendTo("body");

var newPara = $("<p />").text("Hello world");

$("<input />")
 .attr({ "type": "text", "id": "sample" })
 .appendTo("#container");
```

Once created, we can cache these objects using variables and reduce the number of requests to the server.

It's worth noting that design patterns are not exclusive to script code; they can be applied to plugins using similar principles. We will cover more design patterns for jQuery plugins in *Chapter 11, Authoring Advanced jQuery Plugins*.

## Advantages and disadvantages of the Builder Pattern

Using a builder pattern won't suit all circumstances; it is worth noting the benefits that can be gained by using it in order to see if these will suit your requirements. These benefits include the following:

- We can construct the markup needed to create objects dynamically within jQuery, without the need to explicitly create each object
- We can cache the markup, which can then be separated from the main functionality and which makes it easier to read the code and reduce requests to the server
- The core markup will remain immutable, but we can apply different functions to it in order to alter values or its appearance
- We can go further and turn our Builder pattern into a state machine or a mechanism to expose public methods or events, while still maintaining private constructor or destructor methods

There are some disadvantages of using the Builder pattern; the key disadvantage is the abuse of the use of chaining, but we should also consider the following:

- There is scope to define markup that can't be easily reused; this means that we may need to create a handful of the variables that contain markup, all of which will take resources that should be used elsewhere.
- Take an example of the following code snippet:

```
var input = new TagBuilder("button")
 .Attribute("name", "property.name")
 .Attribute("id", "property_id")
 .Class("btn btn-primary")
 .Html("Click me!");
```

The use of the Builder pattern allows actions to be chained, provides a consistent API, and follows the Builder pattern. However, the main drawback of this pattern is that it makes the code harder to read and, therefore, harder to debug.

We've explored a number of different design pattern types at a conceptual level; for some, it may still prove difficult to relate this back to what we know as the jQuery Core.

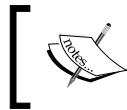
The beauty though is that jQuery uses these patterns throughout—to help put some of what you've learned into practice, let's take a moment to examine the core library and see some examples of how these patterns are used internally.

## Exploring the use of patterns within the jQuery library

Now, you're probably thinking: I'm still not sure how these patterns relate to my work. Right?

Thought so. Throughout this chapter, we've spent time examining some of the more commonly used patterns, as a means of going back to basics; after all, the secret of improving oneself is not just through writing code!

The key point here is that if you spend time developing with jQuery, then you are already using design patterns; to help reinforce what you learned, let's take a look at a few examples from within the jQuery library itself:



For the purposes of this demo, I've used jQuery 2.1.1; if you use a different version, then you may find that some of the line numbers have changed.

1. Start by opening up a copy of `jquery.js` within a text editor of your choice—we'll begin with the classic `document.ready()` function, which uses the Façade pattern and is run from this function at or around line **3375**, as shown in the following screenshot:

```
3374 // Handle when the DOM is ready
3375 ready: function(wait) {
3376
3377 // Abort if there are pending holds or we're already ready
3378 if (wait === true ? --jQuery.readyState : jQuery.isReady) {
3379 return;
3380 }
3381
3382 // Remember that the DOM is ready
3383 jQuery.isReady = true;
3384
3385 // If a normal DOM Ready event fired, decrement, and wait if need be
3386 if (wait !== true && --jQuery.readyState > 0) {
3387 return;
3388 }
3389
3390 // If there are functions bound, to execute
3391 readyList.resolveWith(document, [jQuery]);
3392
3393 // Trigger any bound ready events
3394 if (jQuery.fn.triggerHandler) {
3395 jQuery(document).triggerHandler("ready");
3396 jQuery(document).off("ready");
3397 }
3398 }
```

2. How many times have you toggled the state of an element in your page? I'm guessing it will be a fair few times; the `toggle` command is a classic example of a Strategy design pattern, where we decide the state of an element, as shown here:

```

6108 toggle: function(state) {
6109 if (typeof state === "boolean") {
6110 return state ? this.show() : this.hide();
6111 }
6112
6113 return this.each(function() {
6114 if (isHidden(this)) {
6115 jQuery(this).show();
6116 } else {
6117 jQuery(this).hide();
6118 }
6119 });
6120 }
6121 });

```

3. Now, I'm sure you've clicked on countless elements or used the `click` event handler, right? I hope so, as it is one of the first event handlers we are likely to have started with when first learning about jQuery. It's also a good example of the Observer pattern. Here's the relevant code in jQuery, from around line 7453:

```

7453 jQuery.each(("blur focus focusin focusout load resize scroll unload click dblclick " +
7454 "mousedown mouseup mousemove mouseover mouseout mousenter mouseleave " +
7455 "change select submit keydown keypress keyup error contextmenu").split(" "), function(i, name) {
7456
7457 // Handle event binding
7458 jQuery.fn[name] = function(data, fn) {
7459 return arguments.length > 0 ?
7460 this.on(name, null, data, fn) :
7461 this.trigger(name);
7462 };
7463 });

```

There are plenty more examples of how design patterns are used within the jQuery core library; hopefully, this shows the benefit of using them within your own code and that they should not be limited to the source code for jQuery itself!

## Summary

Phew! We certainly covered a lot of theory on design patterns; let's take a breather and recap what you've learned throughout this chapter.

We kicked off with an introduction to what design patterns are and how they came about; we then moved on to exploring the benefits of using them and why we should consider using them within our projects.

Next up came a look at the structure of a design pattern, where we broke down a typical design into its different elements and saw what role each element plays in the scheme of the design. We also looked at how to categorize design patterns into different types, namely Creational, Structural, and Behavioral.

We then moved on to take a look at a number of common design patterns, where we went through what each type does and examined some examples of how we will use them. We then looked at the benefits and drawbacks of each of the design patterns covered throughout this chapter, before finishing up with a look at how some of these patterns are actually used within the jQuery library itself, and not just within our own code.

I think that's enough theory for now; let's move on and get practical. In the next chapter, we'll see how to take your form development skills up a notch with some techniques to master form development.

# 4

## Working with Forms

How many times have you bought products online, from outlets such as Amazon? I bet the answer is a fair few times over the years—after all, you can't go into a bookstore late at night, peruse the books, and make a choice, without worrying about the store's closing time or knowing whether you will find a particular book.

Building forms for online sites is arguably one of the key areas where you are likely to use jQuery; the key to its success is ensuring that it validates correctly, as a part of offering a successful user experience.

Throughout this chapter, we're going to go back to the basics a little and delve into some of the techniques that we can use to validate forms, using a mix of HTML and jQuery validation tricks. You'll also see that creating successful forms does not require a lot of complex code, but that the process is equally about ensuring that we have considered the form's functionality requirements at the same time.

Over the next few pages, we'll cover a number of topics, as follows:

- Exploring the need for validation
- Adding form validation using regular expressions
- Developing a plugin architecture for validation
- Creating an advanced contact form using jQuery/AJAX
- Developing an advanced file upload form using jQuery

Are you ready to get started? Let's get going...before we start though, I recommend that you create a project folder. For the purpose of this chapter, I will assume that you have done so and that it is called `forms`.

## Exploring the need for form validation

There are different ways to improve the usability of a form, but validation is arguably one of the most important facets that we should consider. How many times have you visited a site and filled in your details only to be told that there is a problem? Sounds familiar, right?

Validating a form is key to maintaining the consistency of information; the form will process the information that has been entered in order to ensure that it is correct. Take an example of the following scenarios:

- If an e-mail address is entered, let's make sure it has a valid format. The e-mail address should include a full stop and contain an @ symbol somewhere in the address.
- Calling someone? What country are they in? Let's make sure that the phone number follows the right format, if we've already set the form to show a specific format of the fields for a chosen country.

I think you get the idea. Now, this might sound as if we're stating the obvious here (and no, I've not lost my marbles!), but all too often, form validation is left until the last stage of a project. The most common errors are usually due to the following reasons:

- **Formatting:** This is where an end user has entered illegal characters in a field, such as a space in an e-mail address.
- **Missing required field:** How many times have you filled out a form, only to find that you've not entered information in a field that is obligatory?
- **Matching error:** This crops up when two fields need to match but don't; a classic example is a password or an email field.

At this stage, you're probably thinking that we're going to get stuck with lots of jQuery, to produce an all-singing, all-dancing solution, right?

Wrong! Sorry to disappoint you, but one mantra I always stick to is the **KISS** principle, or **Keep It Simple, Stupid!** This is not meant as a reflection on anyone, but it is just a way to make our designing lives a little easier. As I've mentioned in an earlier chapter, I believe mastering a technology such as jQuery is not always about the code we produce!

These are the key elements in form validation:

- Tell the user that they have a problem on the form
- Show the user where the problem is
- Show them an example of what you're expecting to see (such as an e-mail address)

Over the next few pages, we're going to take a look at how to add validation to a form and how we can reduce (or eliminate) the most common errors. We'll also work on the use of colors and proximity to help reinforce our messages. However, before we can validate, we need something to validate, so let's knock up a quick form as a basis for our exercises.

## Creating a basic form

As with all projects, we need to start somewhere; in this instance, we need a form that we can use as a basis for adding validation from the various examples given in this chapter.

In the code download that accompanies this book, look for and extract the `basicform.html` and `basicform.css` files to your project folder; when you run `basicform.html`, it will look something similar to this screenshot:

A screenshot of a web page titled "A simple contact form". The form consists of several input fields: "Name:" with a text input field, "Email:" with a text input field, "Website:" with a text input field, and a large "Message:" text area. Below these fields is a "Submit Form" button.

If we take a look at the markup used, we can see that it isn't anything new; it contains standard HTML5 fields that we will use when creating contact forms, such as text fields or text areas:

```
<form class="contact_form" action="" method="post">
 name="contact_form">

 <label for="name">Name:</label>
 <input type="text" name="username" required>

 <label for="name">Email:</label>
```

```
<input type="email" name="email" required>

<button class="submit" type="submit">Submit Form</button>
</form>
```

The key thing here though is that our example doesn't contain any form of validation—it leaves us wide open to abuse of the rubbish in, rubbish out, where users can enter anything and we receive submitted forms that are—well—rubbish! In this instance, when you click on **Submit**, all that you'll see is this screenshot:

A screenshot of a web form with three fields: Name, Email, and Website. The Name field contains "Alex". The Email field is empty and has a red border, indicating it is required. The Website field has a red border and displays the message "Please fill out this field." A cursor arrow points to the error message.

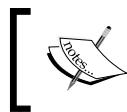
Not great, is it? Most desktop browsers will accept any content if the required tag is used without some of the validation—as long as it has something, the form will be submitted. The exception to this rule is Safari, which won't display the pop-up notice shown in our screenshot.

I'm sure we can do better, but probably not the way you're expecting to see...intrigued?

## Starting with simple HTML5 validation

The great thing about form validation is that it can be easy or complex to fix—it all depends on the route we take to solve the issues.

The key point here is that we *can* use jQuery to provide form validation; this is a perfectly adequate solution that will work. However, for the simple validation of fields, such as names or e-mail addresses, there is an alternative: HTML5 validation, which uses the HTML5 constraint validation API.



The constraint validation API makes use of HTML5 attributes such as `min`, `step`, `pattern`, and `required`; these will work in most browsers, except Safari.

Before I explain the logic within this madness, let's take a quick look at how to modify our demo in order to use this form of validation:

1. Open up a copy of the `basicform.html` file in your usual text editor and then look for this line:

```

<label for="name">Name:</label>
<input type="text" name="username" required>

```

2. We need to add the pattern that will be used as a check for our validation, so go ahead and modify the code as indicated:

```

<label for="name">Name:</label>
<input id="name" name="username" value="" required="required"
pattern="[A-Za-z]+\s[A-Za-z]+" title="firstname.lastname">

```

3. We can make a similar change to our `email` field in order to introduce HTML5 validation; first, look for these lines:

```

<label for="email">Email:</label>
<input type="email" name="email" id="email" required=
"required">

```

4. Go ahead and modify the code as indicated, to add the HTML validation for `email`:

```

<label for="email">Email:</label>
<input type="email" name="email" id="email"
required="required" pattern="[^@]*@[^\s@]*\.[a-zA-Z]{2,}" title="test@test.com">

```

5. Save the file as `basicvalidation.html`; if you preview the results in a browser, you can immediately see a change:

A simple contact form

Name: Alex

Email: Please match the requested format: firstname lastname.

Website:

This is already an improvement; while the text is not very user-friendly, you can at least see that the form expects to see a **firstname lastname** format and not just a forename, as indicated. A similar change will also appear in **Email**, when you press **Submit** to validate your form.

If you look at the code carefully, you may notice that I've switched to using the `required="required"` tags, in place of just `required`. Either format will work perfectly well – you may find that using the former tag is needed, if any inconsistencies appear when you are just using `required` within your browser.

## Using HTML5 over jQuery

Now that we have a form that validates the `name` and `email` fields using HTML, it's time to make good on my promise and explain the logic in my madness.

In some instances, it is often tempting to simply revert to using jQuery in order to handle everything. After all, if we're already using jQuery, why reference another JavaScript library?

This seems like a logical approach to take, if it weren't for these two little issues:

- Using jQuery adds an overhead to any site; for simple validation, this can be seen as an overkill with little return.
- If JavaScript is turned off, then it may result in either the validation failing to operate or errors being displayed on the screen or in the console logs. This will affect user experience, as the visitor will struggle to submit a validated form, or worse, simply leave the site, which might result in lost sales.

A better approach is to consider the use of HTML5 validation for standard text fields and reserve the use of jQuery for more complex validation, as we will see later in this chapter. The benefit of this approach is that we will be able to complete some limited validation, reduce the reliance on jQuery for standard fields, and use it in a more progressive enhancement capacity.

With this in mind, let's move on and start taking a look at the use of jQuery to enhance our forms further and provide more complex validation checks.

## Using jQuery to validate our forms

In some cases, using HTML5 validation will fail if an input type used is not supported in that browser; this is the time when we need to revert to using JavaScript, or in this case jQuery. For example, date as an input type is not supported in IE11, as shown here:

```
<input type="date" name="dob" />
```

This is how the preceding code will be rendered:

```
<input type="text" name="dob" />
```

The trouble is that with the type falling back to text, browsers will not correctly validate the field. To get around this, we can implement a check using jQuery—we can then start adding some basic validation using jQuery, which will override the existing native HTML checks made in the browser.

Let's take a look at how we can achieve some of this in practice, with a simple demo, as follows:

1. Open up a copy of `basicform.html` from the code download that accompanies this book.
2. In the `<head>` section, add a link to jQuery along with a link to your validation script:

```
<script src="js/jquery.js"></script>
<script src="js/basicvalidation.js"></script>
```

3. Save the file as `basicvalidation.html`. In a new file, add the following code—this performs a check to ensure that you are only validating the `email` field:

```
$(document).ready(function () {
 var emailField = $("#email");
```

```
 if (emailField.is("input") && emailField.prop("type") ===
 "email") {
 }
});
```

4. Immediately before the closing }), let's add in the first of two functions; the first function will add a CSS hook to allow you to style in the event of a success or failure:

```
emailField.on("change", function(e) {
 emailField[0].checkValidity();
 if (!e.target.validity.valid) {
 $(this).removeClass("success").addClass("error")
 } else {
 $(this).removeClass("error").addClass("success")
 }
});
```

5. The keen-eyed amongst you will spot the addition of two CSS style classes; we need to allow this in our style sheet, so go ahead and add these lines of code:

```
.error { color: #f00; }
.success { color: #060; }
```

6. We can now add the section function, which alters the default message shown by the browser to show custom text:

```
emailField.on("invalid", function(e) {
 e.target.setCustomValidity("");
 if (!e.target.validity.valid) {
 e.target.setCustomValidity("I need to see an email address
here, not what you've typed!");
 }
 else {
 e.target.setCustomValidity("");
 }
});
```

7. Save the file as `basicvalidation.js`. If you now run the demo in a browser, you can see that the text changes to green when you add a valid e-mail address, as shown in this screenshot:

The screenshot shows a simple web form with two text input fields. The first field is labeled "Name:" and contains the value "Alex Libby". The second field is labeled "Email:" and contains the value "alex@test.com". Both fields have a light blue border, indicating they are active or valid.

- If you refresh your browser session and don't add an e-mail address this time, you will get a custom e-mail address error instead of the standard one offered by the browser, as shown in the following screenshot:

The screenshot shows a simple web form with three fields: 'Email:', 'Website:', and 'Message:'. The 'Email:' field is highlighted with a red border and contains a tooltip message: 'I need to see an email address here, not what you've typed!'. The other two fields ('Website:' and 'Message:') are empty and have no visible validation errors.

Using a little jQuery in this instance has allowed us to customize the message shown – it's a good opportunity to use something a little more user friendly. Note that the default messages given with standard HTML5 validation can be easily...improved!

Now that you've seen how we can change the message that is displayed, let's focus on improving the checks that the form makes. The standard HTML5 validation checks won't be enough for all instances; we can improve them by incorporating checks using regex checks in our code.

## Validating forms using regex statements

So far, you've seen some of the commands that you can use to validate forms using jQuery, and how you can limit your checks to specific field types (such as e-mail addresses) or override the error message displayed on the screen.

The code will fail though, without some form of validation template that we can use to check – the keen-eyed amongst you may have noticed this, in our `basicvalidation.html` demo:

```
pattern = "[^ @]*@[^ @]*\.[a-zA-Z]{2,}";
```

The `pattern` variable is used to define a regular expression or a **regex** statement. Put simply, these are single-line statements that dictate how we should validate any entries in our form. These are not unique to query though; they can be equally used with any scripting language, such as PHP or plain JavaScript. Let's take a moment to look at a few examples in order to see how this one works:

- `[^ @]*`: This statement matches any number of characters that are not an @ sign or a space
- `@`: This is a literal
- `\.`: This is a literal

- `[a-zA-Z]`: This statement indicates any letter, either uppercase or lowercase
- `[a-zA-Z]{2,}`: This statement indicates any combination of two or more letters

If we put this together, the pattern regex translates to an e-mail with any set of characters, save for an @ sign, followed by an @ sign that is then followed by any set of characters except an @ sign, a period, and finally at least two letters.

Okay, enough of theory; let's get coding! We're going to work through a couple of examples, starting with a modification to the e-mail validation and then develop the code to cover validation for website addresses.

## Creating a regex validation function for e-mails

We've already used a regex to validate our `email` address field; while this works well, the code can be improved. I'm not a keen fan of including the validation check within the event handler; I prefer to hive it off into a separate function.

Thankfully, this is easy to correct; let's sort that out now by performing the following steps:

1. We'll start by opening up the `basicvalidation.js` file and adding a helper function immediately before the `emailField.on()` event handler:

```
function checkEmail(email) {
 pattern = new RegExp("[^@]*@[^@]*\\.[a-zA-Z]{2,}");
 return pattern.test(email);
}
```

2. This function handles the validation of e-mail addresses; in order to use it, we need to modify the `emailField.on()` handler, as follows:

```
emailField.on("invalid", function(e) {
 e.target.setCustomValidity("");
 email = emailField.val();
 checkEmail(emailField);
 if (!e.target.validity.patternMismatch) {
 e.target.setCustomValidity("I need to see an email address
here, not what you've typed!");
 }
})
```

If we save our work and then preview it in a browser, we should see no difference in the validation process; we can be rest assured that the validation check process has now been separated into an independent function.

## Taking it further for URL validation

Using the same principles as those used in the previous example, we can develop a similar validation check for the `urlField` field. It's a simple matter of duplicating the two `emailField.on()` event handlers and the `checkEmail` function to produce something similar to what is shown in the following screenshot:

The screenshot shows a simple web form with two fields. The first field is labeled "Email:" and contains the value "alex@test.com". The second field is labeled "Website:" and contains the value "http://www.test.com". Both values are displayed in green, indicating they are valid entries.

Using the code we've already produced, see whether you can create something that validates the website URL entry using this regex:

```
/^(https?:\/\/)?([\da-z\.-]+\.){1,}([a-zA-Z]{2,6})([\w\.-]*\/?$/
```

If your code works, it should produce an error message similar to the one shown in this screenshot:

The screenshot shows a web form with three fields. The first field is labeled "Email:" and contains the value "alex@test.com". The second field is labeled "Website:" and contains the value "test.com", which is enclosed in a red rectangular box, indicating it is an invalid entry. Below the website field, a message box displays the error message: "I need to see a valid website URL here, not what you've typed!"

Hopefully, you've managed to use the code we've produced so far—if you're stuck, there is a working example in the code download that accompanies this book.

So, assuming that we have something that works, has anyone spotted problems with our code? There are definitely some issues that we need to fix; let's go through them now:

- Notice that the feedback isn't 100 percent dynamic? In order to make our code recognize a change from an error to a successful entry, we need to refresh our browser window—this is not ideal at all!
- We're duplicating a lot of code within our jQuery file—architecturally, this is bad practice and we can definitely improve on what has been written.

Instead of duplicating the code, let's completely rework our jQuery into a quick plugin; architecturally, this will get rid of some of the unnecessary duplication and make it easier for us to extend the functionality with minimal changes. It won't be perfect—this is something we will correct later in the chapter—but it will produce a more efficient result than our present code.

## Building a simple validation plugin

Until now, our examples have been based around individual fields, such as an e-mail address or a website URL. The code is heavily duplicated, which makes for a bloated and inefficient solution.

Instead, let's completely flip our approach and turn our code into a generic plugin. We'll use the same core process to validate our code, depending on the regex that has been set within the plugin.

For this next exercise, we'll use a plugin produced by Cedric Ruiz. Although it is a few years old, it illustrates how we can create a single core validation process that uses a number of filters to verify the content entered in our form. Let's make a start by performing the following steps:

1. From the code download that accompanies this book, extract copies of the `quickvalidate.html`, `info.png`, and `quickvalidate.css` files and save them in your project folder.
2. Next, we need to create a plugin. In a new file, add the following code, saving it as `jquery.quickvalidate.js`, within the `js` subfolder of your project area:

```
$.fn.quickValidate = function() {
 return this;
};
```

3. You need to start adding functionality to your plugin, beginning with caching the form and the input fields; add this immediately before the `return this` statement in your plugin:

```
var $form = this, $inputs = $form.find('input:text,
input:password');
```

4. Next up comes the filters that dictate how each field should be validated and the error message that should be displayed when validation fails, as shown here:

```
var filters = {
 required: {
 regex: /.+/,
```

```
 error: 'This field is required'
 },
 name: {
 regex: /^[A-Za-z]{3,}$/,
 error: 'Must be at least 3 characters long, and must only
contain letters.'
 },
 pass: {
 regex: /(?=.*\d)(?=.*[a-z])(?=.*[A-Z]).{6,}/,
 error: 'Must be at least 6 characters long, and contain at
least one number, one uppercase and one lowercase letter.'
 },
 email: {
 regex: /^[\w\-\.\+]+@[a-zA-Z0-9\.\-]+\.[a-zA-Z0-9]{2,4}$/,
 error: 'Must be a valid e-mail address (user@gmail.com)'
 },
 phone: {
 regex: /^[2-9]\d{2}-\d{3}-\d{4}$/,
 error: 'Must be a valid US phone number (999-999-9999)'
 }
};
```

5. We now come to the validation process, which is where the magic happens. Go ahead and add the following code, immediately below the filters:

```
var validate = function(klass, value) {

 var isValid = true, f, error = '';
 for (f in filters) {
 var regex = new RegExp(f);
 if (regex.test(klass)) {
 if (!filters[f].regex.test(value)) {
 error = filters[f].error;
 isValid = false;
 break;
 }
 }
 }
 return {isValid: isValid, error: error}
};
```

6. If your code correctly identifies an error, you need to inform the user; otherwise, they will be left in the dark as to why the form does not appear to be submitted correctly. Let's fix this now by adding in a function to determine what happens if the validation test fails, as follows:

```
var printError = function($input) {
 var klass = $input.attr('class'),
 value = $input.val(),
 test = validate(klass, value),
 $error = $('' + test.error + ''),
 $icon = $('<i class="error-icon"></i>');
 $input.removeClass('invalid').siblings('.error, .erroricon').remove();
 if (!test.isValid) {
 $input.addClass('invalid');
 $error.add($icon).insertAfter($input);
 $icon.hover(function() {
 $(this).siblings('.error').toggle();
 });
 }
};
```

7. We've determined what will happen when the validation process fails but haven't put anything in place to call the function. Let's fix this now by adding in the appropriate call, based on whether the field is marked as being required, as shown here:

```
$inputs.each(function() {
 if ($(this).is('.required')) {
 printError($(this));
 }
});
```

8. If the content changes in our field, we need to determine whether it is valid or invalid; this needs to take place when entering text, so let's do that now, using the keyup event handler:

```
$inputs.keyup(function() {
 printError($(this));
});
```

9. Finally, we need to prevent submission if errors are found in our form:

```
$form.submit(function(e) {
 if ($form.find('input.invalid').length) {
 e.preventDefault();
```

```

 alert('There are errors on this form - please check...');

 }

 return false;
}) ;

```

10. Save your work; if all is well, you should see the form validate when previewing the results of your work in a browser:

The screenshot shows a simple HTML form with four fields: Name, Password, E-Mail, and Phone. The Name field contains "Alex". The Password field contains "\*\*\*\*\*" and is highlighted with a yellow background, indicating it's invalid. A tooltip box appears over the Password field with the text: "Must be at least 6 characters long, and contain at least one number, one uppercase and one lowercase letter.". The E-Mail and Phone fields are empty. Below the form, a red message box displays the error message: "There are errors on this form - please check...".

At this stage, we have a working plugin, where we've refactored the core validation code into a single set of processes that can be applied to each field type (using the appropriate filter).

However, we can do better than this; the following are some issues that we can address to take our code even further:

- Although we've refactored the code into a single set of core validation processes, the filters still form a part of the core code. While it is easy to expand on the different types of filters, we are still limited to either text or password field types. Adding any of the standard HTML5 field types, such as `url` or `email`, will result in an error, as the pseudo-type is not supported within jQuery.
- From an architectural perspective, it is preferable to keep the validator filters outside the core plugin; this helps to keep the validator lean and free from code that isn't required for our purpose.
- Our code doesn't allow for any features such as localization, setting a maximum length, or the validation of form objects, such as checkboxes.

We can spend lots of time developing our plugin so that it takes a more modular approach, but is it worth the effort? There are literally dozens of form validation plugins available for use; a smarter move will be to use one of these plugins:

- The core validation process is tried and tested, which eliminates the need to worry about whether our fields will validate correctly. Developing any form of validator plugin that works on more than just a few fields is notoriously tricky to get right; after all, what do we look or don't look to validate? Different languages? Different formats for postal or zip codes, for example?
- Most plugins will have some form of architecture to allow the addition of custom validators, which supplement those included as standard—examples include the use of languages, specific number formats, or odd/even numbers. We will make full use of this later in this chapter in order to add some custom validators to our demo.
- Using an existing plugin allows you to concentrate on providing the functionality that is specific to your environment and where you can add the most value—after all, there is no point in trying to add valid where others have already done the work for us, right?

With this in mind, let's move on and take a look at how to use an existing plugin. Most plugins nowadays have some form of modular architecture that allows you to easily customize it and add additional custom validators; after all, why spend time reinventing the wheel, right?

## Developing a plugin architecture for validation

Throughout this chapter, we've worked with a variety of HTML5 and jQuery techniques to validate our forms. In the main, they have worked well, but their simplistic nature means that we will easily outgrow their usefulness very quickly.

To really take advantage of all that is possible with form validation, it makes sense to move away from simply trying to validate fields to using an existing plugin that takes care of the basic validation process and allows you to concentrate on customizing it and on ensuring that you provide the right functionality for your form.

Enter jQuery Form Validator. This plugin, created by Victor Jonsson, has been around for a number of years, so it is tried and tested; it also contains the modular architecture that we need to customize the checks we will provide within our form. Let's take a look at the validator in action.



The original plugin and associated documentation are available at <http://formvalidator.net/>.

## Creating our basic form

Before we start to add custom validator plugins to our code, we need a basic form to validate. For this, we'll base the markup on a modified version of the form created in `basicvalidation.html`, from an earlier section in this chapter.

Let's get our basic form working, with standard validation in place. To do this, perform the following steps:

1. We'll start by extracting copies of the `formvalidator.html` and `formvalidator.css` files from the code download that accompanies this book. Save the HTML file in the root of your project folder and the CSS file in a `css` subfolder.
2. In a new file, add the following lines of code, saving it as `formvalidator.js` in a `js` subfolder of your project area:
 

```
$(document).ready(function() {
 $.validate();
});
```
3. This is all that is needed to get started with the Form Validator plugin; if you preview the form in a browser, you should see the following screenshot—if you enter a valid name and e-mail address but omit the website URL:

A simple contact form

Name:	Alex
Email:	test@test.com
Website:	
The answer you gave was not a correct URL	
Message:	
<input type="button" value="Submit Form"/>	

Now that our form is ready, let's really start developing some of the validators used within the form, beginning with a new validator for the `name` field.

## Creating custom validators

So far, our form has relied on using standard HTML5 techniques to validate it; this will work for most requirements, but there is a limit to what it can do. Enter jQuery; we can use the power of FormValidator to create our own custom validators so that we can tailor them to match our own requirements.

The key part of creating custom validators is the `$.formutils.addValidator` configuration object; FormValidator handles the basic plugin architecture, which leaves you to add values by designing the right checks for your form.

Over the next few pages, we're going to work through two basic examples:

1. We'll start by creating our custom validator; in the usual text editor of your choice, add the following code:

```
$.formUtils.addValidator({
 name : 'user_name',
 validatorFunction : function(value, $el, config, language,
 $form) {
 return (value.indexOf(" ") !== -1)
 },
 errorMessage : 'Please enter your full name',
});
```

2. Save the file as `user_name.js`, within the `js` subfolder of your project area. Open up the `formvalidator.js` file that you created in the previous section and alter it as shown here:

```
$(document).ready(function() {
 $.formUtils.loadModules('user_name');
 $.validate({
 modules: 'user_name'
 });
});
```

- Although you've added the validation rule to the validator, you need to activate it from within your HTML markup, as follows:

```
<div class="form-group">
 <label class="control-name" for="name">Name: *</label>
 <input name="username" id="username" datavalidation="user_name">
</div>
```

- If all works well, you will see the effects of using your custom validator when you preview the form in a browser and press the **Submit** button, as shown in the following screenshot:

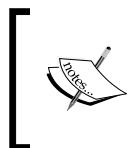
The screenshot shows a web form with three fields: Name, Email, and Website. Each field has a red border and a red error message below it.

- Name:** The input field is empty. The error message is "Please enter your full name".
- Email:** The input field contains an invalid email address. The error message is "You have not given a correct e-mail address".
- Website:** The input field contains an invalid URL. The error message is "The input value is not a correct URL".

At this stage, you can simply leave it with this custom validator in place, but I think there is scope for more – what about e-mail addresses?

Standard HTML5 validation will work out if the e-mail address given is in a suitable format, such as ensuring that it has an @ sign, a decimal point after the domain name, and that the domain suffix is valid. It won't, however, prevent users from submitting forms with certain types of addresses, such as `www.hotmail.com` (or now `www.outlook.com`).

At this point, it is worth noting that e-mail validation using regexes can open up a minefield of problems, so step carefully and test thoroughly – how do you validate against `mail+tag@hotmail.com`, for example? This is a perfectly valid address but most regexes will fail...



A useful discussion on why using regexes can actually do more harm than good is available at <http://davidcelis/blog/2012/09/06/stop-validating-email-addresses-with-regex/>.

In our example, we'll add a simple check to prevent Hotmail, Gmail, or Yahoo! e-mail addresses from being used; let's take a look at how we can do that:

1. In a text editor, add the following code to a new file, saving it as `free_email.js` within your `js` subfolder:

```
$.formUtils.addValidator({
 name : 'free_email',
 validatorFunction : function(value, $el, config, language,
 $form) {
 varemailName = /^([\w-\.]+@[?!gmail.com](!yahoo.com)
(!hotmail.com)([\w-]+\.)+[\w-
]{2,4})?$/;
 return (emailName.test(value))
 },
 errorMessage : 'Sorry - we do not accept free email accounts
such as Hotmail.com'
});
```

2. Now that your `free_email` validator is in place, you need to call it when validating your form; to do this, revert to the `formvalidator.js` file you had opened in the previous exercise and amend the code, as shown here:

```
$(document).ready(function() {
 $.formUtils.loadModules('free_email');
 $.validate({ modules: 'free_email'});
});
```

3. The final step in the exercise is to activate the custom validator from the HTML markup – remember how we changed it in the previous exercise? The same principle applies here too:

```
<div class="form-group">
 <label class="control-name" for="email">Email: *</label>
 <input type="text" name="email" id="email" datavalidation="free_
 email">
</div>
```

4. Save both the `formvalidator.js` and `formvalidator.html` files; if you preview the results of your work, you can clearly see your custom message appear if you've entered an invalid e-mail address, as shown in the following screenshot:

**A simple contact form**

Name:	<input type="text"/>
Email:	<input type="text" value="alex.libby@hotmail.com"/>
Sorry - we do not accept free email accounts such as Hotmail.com	

Now, the observant amongst you will spot that we're loading one validator at a time; I am sure that you are wondering how we can load multiple validators at the same time, right?

No problem, we already have the validator files in place, so all we need to do is modify our validator object so that it loads both the modules. Let's take a quick look at how we can modify our validator object:

1. Open up a copy of the `formvalidator.js` file and alter the code as shown here:

```
$.formUtils.loadModules('free_email, user_name');
$.validate({
 modules: 'free_email, user_name',
});
```

That's all that you need to do. If you save the file and preview the results in a browser, you will find that it validates both the `name` and `email` fields, as illustrated in the previous two exercises.

This opens up a world of opportunities; in our two examples, we've created reasonably simple validators but the principles are the same, no matter how complex or simple our validators are.

If you want to learn more about how to create custom validators, then it is worth reading the documentation at <http://formvalidator.net/index.html#custom Validators>. We can then combine the basic principles of creating modules with regex examples such as those shown at <http://www.sitepoint.com/jquery-basic-regex-selector-examples/>, to create some useful validator checks.

Let's move on and take another look at a useful part of the FormValidator plugin—we all don't speak the same language, do we? If we did, life would be boring; instead, you should consider localizing your validation messages so that international visitors to your site can understand where there is a validation issue and know how to fix it.

## Localizing our content

In this modern age of working online, there may be instances where it will be useful to display messages in a different language—for example, if most of your visitors speak Dutch, then there will be value in overriding the standard messages, with equivalents in the Dutch language.

While it requires some careful thought and planning, it is nevertheless very easy to add language support; let's run through how to do so:

1. For this exercise, you need to modify the validator object. In the `formvalidator.js` file, add this code immediately after the `document.ready()` statement:

```
var myLanguage = {
 badUrl: "De ingangswaarde is geen correcte URL"
};
```

2. We need to reference the change in language, so go ahead and add this configuration line to the validator object:

```
$.validate({
 modules: 'free_email, user_name',
 language: myLanguage
});
```

3. Save the file. If you preview the results in a browser, you can see that the error message is now displayed in Dutch, as shown here:

A simple contact form

Name:

Email:

Website:  De ingangswaarde is geen correcte URL

4. We're not limited to Dutch; here's the same code, but with an error message in French:

```
varmyLanguage = { badUrl: "La valeur d'entrée n'est pas une URL correcte" };
```

This is a quick and easy way to ensure that visitors to your site understand why your form hasn't validated and how they can fix it. It is worth noting though that the message set is displayed irrespective of the regional settings on your PC or mobile device; it is recommended that you check any analytics logs to confirm the region or country your visitors come from before changing the language in use on your form's messages.

## Centralizing our error messages

Before we wrap up development on our form, there is one more piece of functionality that we can look at in order to add it to our form.

So far, any validation error message that is displayed has been against each individual field. This works, but it means that we don't have an immediate way of telling which fields may have failed validation. Sure, we can scroll through the form, but I'm lazy; why scroll down a long form if we can alter our code to display the errors at the top, right?

Absolutely, doing this is a piece of cake with FormValidator; let's go through what is required now:

1. Open a copy of the `formvalidator.js` file and alter the validator object as shown here; we set the `errorMessagePosition` property to `top` and the `validatorOnBlur` property to `false` in order to display messages at the top of the form:

```
$.validate({ modules: 'user_name, free_email', validateOnBlur
 : false, errorMessagePosition : 'top', language: myLanguage
});
```

2. If you were to run the form now, any error messages that have been set will display at the top, but they won't look pretty. Let's fix this now, with some minor changes to our style sheet:

```
div.form-error {
 font-size: 14px;
 background: none repeat scroll 0 0 #ffe5ed;
 border-radius: 4px; color: #8b0000;
 margin-bottom: 22px; padding: 6px 12px;
 width: 88%; margin-left: 0px; margin: 10px;
}
```

- Now, let's run the form in a browser; if all went well, you will see the errors at the top of the form, correctly formatted. The following screenshot shows what might appear if you were to not fill out the website URL; note that our code still shows the message in Dutch from the previous example:

The screenshot shows a modal dialog titled "Form submission failed!" with a pink background. It contains a single bullet point: "• De ingangswaarde is geen correcte URL". Below the dialog, the form fields are shown: Name: \* (green field with checkmark), Email: \* (green field with checkmark), and Website: \* (red field with exclamation mark). The red field has a tooltip below it: "Sorry - we do not accept free email accounts such as Hotmail.com".

At this point, we've covered a number of topics related to validation using jQuery. We're going to move on and take a look at a couple of example forms in action. Before we do so, we need to cover some final tweaks as part of wrapping up development.

## Wrapping up development

When previewing the last exercise, the more observant will have spotted that some of the styles appear to be missing. There is a good reason for this; let me explain.

As a minimum, we can provide messages to indicate success or failure. This will work but it isn't great; a better option is to provide some additional styling to really set off our validation:

The screenshot shows a "A simple contact form" dialog with a brown header. A small note at the top right says "\* denotes a required field". The form fields are: Name: \* (green field with checkmark), Email: \* (red field with exclamation mark and tooltip: "Sorry - we do not accept free email accounts such as Hotmail.com"), and Website: \* (red field with exclamation mark and tooltip: "De ingangswaarde is geen correcte URL"). Below the fields is a "Message:" label followed by a large text input area. At the bottom is a "Submit Form" button.

This is easy to do, so let's make a start by performing the following steps:

1. Open up the `formvalidator.css` file and add the following lines of code:

```
.has-error, .error { color: #f00; }
.has-success, .valid { color: #060; }
.error { background-image: url(..../img/invalid.png); background-position: 98%; background-repeat: no-repeat; background-color: #ff9a9a; }
.valid { background-image: url(..../img/valid.png); background-position: 98%; background-repeat: no-repeat; background-color: #9aff9a; }
```
2. We need to add two icons to the `img` subfolder in our project area—for this, I've used the red cross and green tick icons available at [https://www.iconfinder.com/icons/32520/accept\\_check\\_good\\_green\\_ok\\_success\\_tick\\_valid\\_validation\\_vote\\_yes\\_icon](https://www.iconfinder.com/icons/32520/accept_check_good_green_ok_success_tick_valid_validation_vote_yes_icon). If you want to use different icons, then you may have to adjust the style rules accordingly.
3. Save `formvalidator.css`. If you preview the results in a browser and enter details in the form, you should see results similar to the screenshot shown at the start of this exercise when you click on **Submit Form**.

Hopefully, you will agree that this looks much better! There is a copy of `formvalidator.css` in the code download that accompanies this book; it contains a few more styles within the form that give it the really polished look that we've seen in this exercise.



If you want to see a working example, which contains the customizations, then extract the `formvalidator-fullexample` JavaScript, CSS, and HTML files from the code download and rename them to `formvalidator.js`, `formvalidator.css`, and `formvalidator.html`, respectively.

## Noting the use of best practices

In each of our examples, we've set the form to display all the fields at once—a key point to consider the user's goals and expectations. What are they trying to achieve? Do we really need to display dozens of fields at once? Or, can we make the form simpler?

Although the focus of this book is naturally on mastering jQuery, it would be foolish to simply concentrate on writing code; we must also give some consideration to the look and feel of the form and allow any visual or functional considerations when building the form and its associated validation.

As a small example, it might be worth considering whether we can use CSS to blur or focus fields, as and when fields become available. We can achieve this using a small amount of CSS to blur or focus those fields, using something similar to the following code:

```
input[type=email], input[type=text] { filter: blur(3px); opacity: .4; transition: all .4s; }

input[type=email]:focus, input[type=text]:focus { filter: none; opacity: 1; }
```

The idea here is to fade out those fields where we have entered something and focus on those fields that we have yet to complete or are about to complete, as shown in the following screenshot:



A small warning: if we are not careful when using this styling, we may appear to effectively disable fields, which will kill the whole point of the exercise! Let's change focus now and switch to a key part of form design: what happens if some browsers don't support the CSS styles we've used throughout this chapter?

## Providing fallback support

Throughout this chapter, we've pushed the boat out in designing forms that will work in most modern browsers. There may be instances though when this won't work; if we still have to cater to nonsupporting browsers (such as iOS7), then we need to provide some form of fallback.

Thankfully, this isn't too much of an issue if we use something such as a Modernizr to provide a graceful degradation by applying the `formvalidation` class on the `html` element. We can then use this to provide a graceful fallback if a browser doesn't support the use (and styling) of pseudo-selectors, such as `:valid` or `:invalid`.



If you want to use a custom version of Modernizr, which will test for form validation support, then go to [http://modernizr.com/download/#-shiv-cssclasses-teststyles-testprop-testallprops-prefixes-domprefixes-forms\\_validation-load](http://modernizr.com/download/#-shiv-cssclasses-teststyles-testprop-testallprops-prefixes-domprefixes-forms_validation-load).

Enough of the theory, let's have some fun! Over the course of the next couple of pages, we're going to have a look at a more complex example, over two exercises. It will be based on a simple contact form to which we will add form upload capabilities – although beware, as there will be a sting in this tail...!

## Creating an advanced contact form using AJAX

In the first part of our complex example, we're going to develop a form that allows us to submit some basic details and that allows the confirmation of this submission to first appear on a form message panel and later by e-mail.

For this exercise, we will need to avail ourselves of a couple of tools, as follows:

- A local web server installed using default settings – options include WAMP (for PC; <http://www.wampserver.de/en>) or MAMP (for Mac; <http://www.mamp.info/en/>). Linux users will most likely already have something available as a part of their distribution.
- The free Test Mail Server tool (for Windows only), available at <http://www.toolheap.com/test-mail-server-tool/>, as e-mailing from a local web server can be difficult to set up, so this brilliant tool monitors port 25 and provides local e-mailing capabilities. For Mac, you can try the instructions provided at <https://discussions.apple.com/docs/DOC-4161>; Linux users can try following the steps outlined at <http://cnedelcu.blogspot.co.uk/2014/01/how-to-set-up-simple-mail-server-debian-linux.html>.
- Access to an e-mail package from the PC or laptop that is being used – this is required to receive e-mails sent from our demo.



Another possible option, if you prefer to go down the cross-browser route, is XAMPP (<https://www.apachefriends.org/index.html>); this includes the Mercury Mail Transport option, so the Test Mail Server tool isn't required if you are working on Windows.

Okay, with the tools in place, let's make a start by performing the following steps:

1. We're going to start by opening up a copy of the code download that accompanies this book and extracting the `ajaxform` folder; this contains the markup, styling, and assorted files for our demo. We need to save the folder into the web server's `www` folder, which (for PC) will usually be `C:\wamp\www`.
2. The markup is relatively straightforward and very similar to what we've already seen throughout this chapter.
3. We need to make one small change to the `mailer.php` file; open it in a text editor of your choice and then look for this line:

```
$recipient = "<ENTER EMAIL HERE>";
```

4. Change `<ENTER EMAIL HERE>` to a valid e-mail address that you can use in order to check whether an e-mail has appeared afterwards.
5. The magic for this demo happens within the `ajax.js` file, so let's take a look at the file now and begin by setting some variables:

```
$(function() {
 var form = $('#ajaxform');
 var formMessages = $('#messages');
```

6. We start the real magic here, when the submit button is pressed; we first prevent the form from submitting (as it's the default action) and then serialize the form data into a string for submission:

```
$(form).submit(function(e) {
 e.preventDefault();
 var formData = $(form).serialize();
```

7. The core of the AJAX action on this form comes next; this function sets the type of request to make, where the content will be sent to and the data to send:

```
$.ajax({
 type: 'POST',
 url: $(form).attr('action'),
 data: formData
})
```

8. We then add the two functions to determine what should happen; the first function deals with the successful submission of our form:

```
.done(function(response) {
 $(formMessages).removeClass('error');
 $(formMessages).addClass('success');
 $(formMessages).text(response);
```

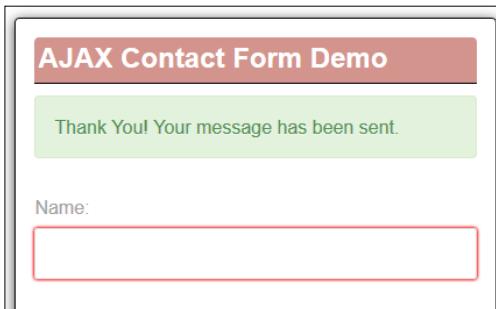
```
$('#name').val('') ;
$('#email').val('') ;
$('#message').val('') ;
})
```

9. Next up comes the function that handles the outcome if form submission fails:

```
.fail(function(data) {
 $(formMessages).removeClass('success');
 $(formMessages).addClass('error');

 if (data.responseText !== '') {
 $(formMessages).text(data.responseText);
 } else {
 $(formMessages).text('Oops! An error occurred and your
message could not be sent.');
 }
});
});
});
```

10. Start the Email Tool. If you preview the form in a browser and fill out some valid details, you should see this screenshot when you submit it:



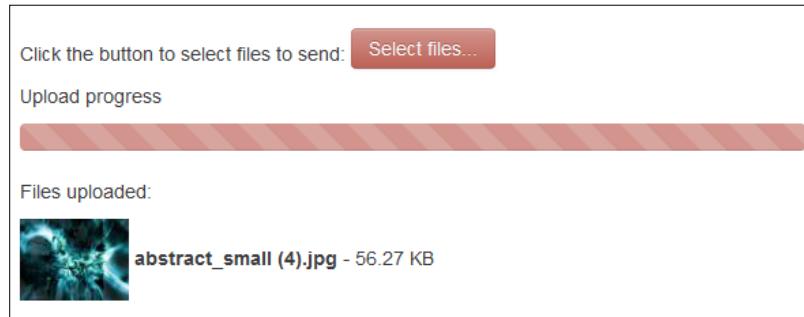
Our form is now in place and able to submit, with the confirmation appearing by e-mail within a few moments. We will revisit the use of AJAX within jQuery in greater depth in the next chapter; for now, let's move on and continue to develop our form.

## Developing an advanced file upload form using jQuery

As one good man said some time ago, "*onwards and upwards!*", it's time to add the second part of our form's functionality, in the form of a file upload option.

Leaving aside the risks that this can present (such as the uploading of viruses), adding a file upload function is relatively straightforward; it requires both client- and server-side components to function correctly.

In our example, we're going to focus more on the client-side functionality; for the purpose of the demo, we will upload files to a fake folder stored within the project area. To give you an idea of what we will build, here's a screenshot of the completed example:



With this in mind, let's make a start by performing the following steps:

1. In a copy of the ajaxform.html file, we need to add some additional links to various JavaScript and CSS files; all the additions are available in the code download that accompanies this book, as shown here:

```
<link rel="stylesheet" href="css/bootstrap.min.css">
<link rel="stylesheet" href="css/styles.css">
<link rel="stylesheet" href="css/fileupload.css">
<script src="js/jquery.min.js"></script>
<script src="js/ajax.js"></script>
<script src="js/jquery.ui.widget.js"></script>
<script src="js/jquery.iframe-transport.js"></script>
<script src="js/jquery.fileupload.js"></script>
<script src="js/uploadfiles.js"></script>
```

2. Next, we need to add some markup to `index.html`; so, in `ajaxform.html`, go ahead and first alter the title as shown here:

```
<div id="formtitle">
 <h2>File Upload Demo</h1>
</div>
<div id="form-messages"></div>
```

3. We now need to add the file upload code; so, immediately after the closing `</div>` tag of the message field, add the following code:

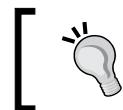
```
<div class="container">
 Click the button to select files to send:

 Select files...
 <input id="fileupload" type="file" name="files[]" multiple>

 <p>Upload progress</p>
 <div id="progress" class="progress progress-success
progress-striped">
 <div class="bar"></div>
 </div>
 <p>Files uploaded:</p>
 <ul id="files">
</div>
```

4. We need to make one small change to one of our jQuery files; in `uploadfiles.js`, look for the line that begins as follows:

```
$('#files').append('url</code> | The URL of the content, for the request. |
| <code>data</code> | The data to be sent to the server. |
| <code>error</code> | This function is called in the event of the request failing – the function will be passed three arguments: the <code>jqXHR</code> object, a string describing the error, and an optional exception object, if one is generated. |
| <code>dataType</code> | This describes the type of data that you're expecting to see returned from the server. By default, jQuery will try to work this out automatically, but it could be one of the following: XML, JSON, script, or HTML. |
| <code>success</code> | A function to be called if the request is successful. |
| <code>type</code> | The type of request to make, for example, 'POST', 'GET' or 'PUT' – the default is 'GET'. |



There are many more options available. For a reminder, it is worth browsing to <http://api.jquery.com/jQuery.ajax/> for more details.



Enough theory – at least for the moment! Let's move on and take a look at developing an example using AJAX and jQuery.

Creating a simple example using AJAX

Before we get stuck in developing code, and pushing the boundaries of what we can do, let's spend a moment understanding what typical AJAX code looks like in action.

In a typical application that relies on importing content, we might come across something akin to the following extract:

```
var jqxhr = $.ajax({
  url: url,
  type: "GET",
  cache: true,
  data: {},
  dataType: "json",
  jsonp: "callback",
  statusCode: {
    404: handler404,
    500: handler500
  }
});
jqxhr.done(successHandler);
jqxhr.fail(failureHandler);
```

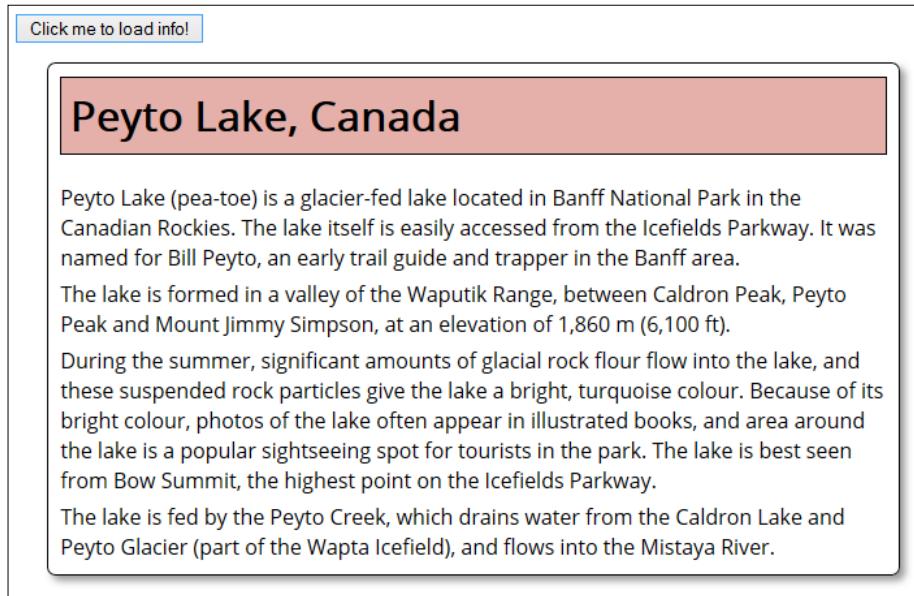
This is a standard configuration object for AJAX-enabled code. Let's take a look at some of these configuration options in a little more detail:

| Option | Comments |
|----------|---|
| url | The URL of the |
| type | Default is GET, but other verbs can be used instead, if required |
| cache | The default is true, but false for 'script' and 'jsonp' datatypes, so must be set on a per case basis |
| data | Any request parameters should be set in the data object |
| datatype | The datatype should be set for future reference |

| Option | Comments |
|------------|---|
| jsonp | Only specify this to match the name of the callback parameter your API is expecting for JSONP requests, which are being made of a server hosted in a different domain |
| statusCode | If you want to handle specific error codes, use the status code mapping settings |

[ There is plenty of documentation on the jQuery Core site – it is well worth reading! A good place to start is with the main `ajax()` object, at <http://api.jquery.com/jquery.ajax/>.]

We can use it to great effect to produce a simple demo, such as displaying information from an XML file, or even plain HTML, as shown in the next screenshot:



Click me to load info!

Peyto Lake, Canada

Peyto Lake (pea-toe) is a glacier-fed lake located in Banff National Park in the Canadian Rockies. The lake itself is easily accessed from the Icefields Parkway. It was named for Bill Peyto, an early trail guide and trapper in the Banff area.

The lake is formed in a valley of the Waputik Range, between Caldron Peak, Peyto Peak and Mount Jimmy Simpson, at an elevation of 1,860 m (6,100 ft).

During the summer, significant amounts of glacial rock flour flow into the lake, and these suspended rock particles give the lake a bright, turquoise colour. Because of its bright colour, photos of the lake often appear in illustrated books, and area around the lake is a popular sightseeing spot for tourists in the park. The lake is best seen from Bow Summit, the highest point on the Icefields Parkway.

The lake is fed by the Peyto Creek, which drains water from the Caldron Lake and Peyto Glacier (part of the Wapta Icefield), and flows into the Mistaya River.

Let's take a look at this demo in more detail:

1. From the code download that accompanies this book, extract copies of the `basicajax.html`, `content.html`, and `basicajax.css` files – place the HTML files into the root of our project folder, and the style sheet into the `css` subfolder.

2. Next, add the following code to a new file, saving it as `basicajax.js` in the `js` sub-folder of our project area:

```
$(document).ready(function() {
});
```

3. Immediately below the declared `$description` variable, add the following helper function to control the rendering of our extracted text on screen:

```
var displaytext = function(data) {
    var $response = $(data), $info = $("#info");
    var $title = $('<h1>').text($response.find(".title")
        .text());
    $info.append($title);
    $response.find(".description").each(function() {
        $(this).appendTo($info);
    });
};
```

4. Next up comes the core of our jQuery code – the call to `$.ajax`. Add the following event handler immediately below the helper function:

```
$('#action-button').click(function() {
    $.ajax({
        url: 'content.html',
        data: { format: 'html' },
        error: function() {
            $('#info').html('<p>An error has occurred</p>');
        },
        dataType: 'html',
        success: displaytext,
        type: 'GET'
    });
});
```

5. If we preview the results in a browser, we can see the content appear when clicking on the button, as shown in the screenshot at the start of this demo.

In this instance, we've created a simple demo. It first references the `content.html` file, using the HTML format to import it to our page. Our jQuery code then pulls the content and assigns it to `$response`, before first extracting the title, then each of the paragraphs, and appending them to the `#info` div.

At this point, it is worth noting that we could have referenced each of those extracted paragraphs individually, using a statement such as the following:

```
var $description1 =
$('<p>').text($response.find(".description:eq(0)").text());
```

This however is an inefficient way to extract the text – we would have to run the code multiple times to reference subsequent values, which places an unnecessary load on our server.

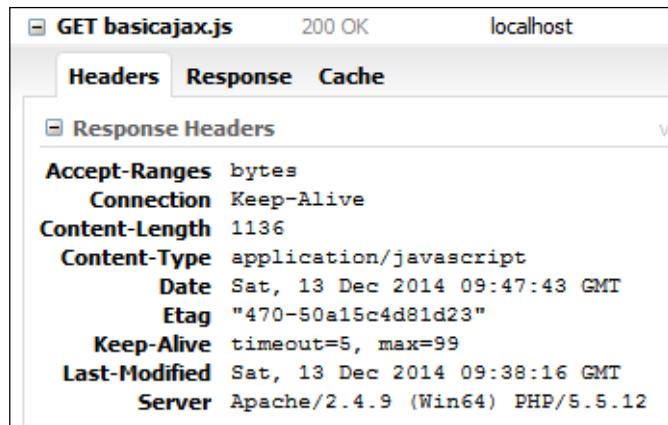
Improving the speed of loading data with static sites

Now that we have seen an AJAX example in action, it may surprise you to learn that the code used isn't *technically* as efficient as it could be, even with the small amount of text that we displayed on screen.

Huh? I hear you ask – surely we can't really improve on such a simple demo, right? Well, strange as it might seem, we can already make one improvement. Let's take a look at some of the tricks we can use to reduce any slowness in our code – not all of them have to do with simply changing our code:

- Reduce the number of AJAX requests – no, I've not lost the plot; improving our code isn't always about making changes to the code itself! If we consider when each AJAX request is made, there may be opportunities to reduce the number, if reordering means we can achieve the same result. For example, if we have AJAX requests being made on a timer, we can set a flag to indicate that AJAX requests should only be performed in changes have been made.
- If we need to fetch content, then it is often more effective to simply use GET, rather than POST – the former simply retrieves content, while the latter will cause a server reaction, such as updating a database record. If we don't need to perform an action, then using GET is perfectly adequate.
- When updating content on a page – make sure you are only updating a small amount; AJAX performance will be affected if our page is set to update a broad sweep of content, rather than a defined section.
- Reduce the amount of data to be transmitted – remember I said there was a change we could make to our code? Here's where we can make it – while we don't need to limit the content we retrieve, we can change from using HTML format to plain text. This allows us to remove the markup tags, thereby reducing our content. We could always go in the completely opposite direction, and switch to using XML, but this wouldn't be without an equal increase in data size!

- We should also check that our server has been properly configured – the two key areas to check are the use of ETags (or Entity Tags), and that the server is set to send the correct expires or Cache-Control headers for the content being served, as shown in the next example:



- In a nutshell, server will not send any response if it detects that ETags for a URL have not changed.



Head over to http://en.wikipedia.org/wiki/HTTP_ETag if you would like to learn more about ETags and how they work in a browser.

- We can further limit the impact of AJAX requests by only creating and destroying the XMLHttpRequest at the right time – if they are only needed at certain instances, then this will have a dramatic effect on AJAX performance. For example, we might only instigate an AJAX request if our code doesn't have an active class:

```
if (!($this).hasClass("active")) {  
    ...perform our ajax request here...  
}
```

- Ensure that your callbacks are set correctly – if our code has been updated, then we need to tell our users as much, and not keep them waiting; after all, the one thing we do not want to do is fall into the trap of callback hell! (Later in this chapter, we will cover this in more detail.)

We can take things even further! One way we can reduce unnecessary calls to the server is by caching content. But – before you say "I know that", I didn't say where!

Yes – the *where* in this instance is key and the *where* is – the `localStorage`. This is built into each browser, and can be used to remove the need to continually hit the server. While the amount you can store varies from browser to browser (it's typically 5 MB, but can be as high as 20 MB), it works using the same principles for each browser – the content must be stored as text, but can include images and text (within reason!).

Intrigued? Using a simple plugin and making some small changes to code, we can quickly implement a workable solution – let's revisit our basic AJAX demo from earlier, and make those changes now.

Using `localStorage` to cache AJAX content

Working with AJAX requires careful consideration – it is important to strike a balance in fetching the right amount of content, at the appropriate points, without making too many unnecessary requests to the server.

We've seen a number of tricks we can use to help reduce the impact of AJAX requests. One of the more adventurous ways is to store content in the `localStorage` area of each browser – we can do this using an AJAX prefilter. The developer Paul Irish has wrapped up the code needed to do this in a plugin, which is available at <https://github.com/paulirish/jquery-ajax-localstorage-cache>.

We're going to use it to alter our `basicajax` demo from earlier. Let's take a look at how we are going to do this:

1. Let's start by extracting a copy of the `basicajax` demo folder from the code download that accompanies this book, and saving it to our project area.
2. Next, we need to download the plugin – this is available at <https://github.com/paulirish/jquery-ajax-localstorage-cache/archive/master.zip>. From the `zip` file, extract `jquery-ajax-localstorage-cache.js`, and save it to the `js` subfolder within `basicajax`.
3. We need to make some changes to our JavaScript and HTML markup. Let's first change the JavaScript. In `basicajax.js`, add the following two lines as shown:

```
localCache: true,  
error: function() {  
    cacheTTL: 1,
```

4. In `basicajax.html`, we need to reference the new plugin, so go ahead and alter the script calls, as shown next:

```
<script src="js/basicajax.js"></script>
<script src="js/jquery-ajax-localstorage-cache.js"></script>
</head>
```

5. If we rerun the demo and click on the button to load the content, we should not see anything different visually; the change will be apparent if we fire up Firebug, switch to the **Net** tab, and then click on **JavaScript**:

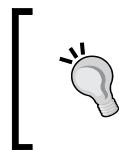
The screenshot shows the Firebug Net tab interface. At the top, there are tabs for Console, HTML, CSS, Script, DOM, Net (selected), and Cookies. Below the tabs, there are buttons for Clear and Persist, and dropdowns for All, HTML, CSS, JavaScript (which is selected), XHR, Images, and Plugins. A table below lists network requests. The first row shows a request for 'GET jquery.min.js' with status '200 OK (BFCache)', domain 'localhost', and size '81.7 KB'. The second row shows a request for 'GET basicajax.js' with status '200 OK', domain 'localhost', and size '1.1 KB'. The IP address is listed as ':1:80'.

URL	Status	Domain	Size	Remote IP
GET jquery.min.js	200 OK (BFCache)	localhost	81.7 KB	
GET basicajax.js	200 OK	localhost	1.1 KB	[:1]:80

6. If we explore further, we can now see signs of our AJAX content being stored within the **localStorage** area of our browser:

The screenshot shows the Firebug Storage tab interface. It displays the contents of the localStorage object. There are two items: 'content.htmlGETformat=html' and 'localStorage'. The 'content.htmlGETformat=html' item is expanded, showing its value as a string of HTML code representing a page about Peyto Lake.

```
localStorage
  ↳ 2 items in Storage content.htmlGETformat=html="<!DOCTYPE html>\r\n<html>\r...ul>\r\n</body>\r\n</html>\r\n",
  content.htmlGETformat=htmlcachetl="1418467206830"
content.htmlGETformat=html
  ↳ <!DOCTYPE html>
    <html>
      <head>
        </head>
      <body>
        <ul id="peyto">
          <li class="title">Peyto Lake, Canada</li>
          <li class="description">Peyto Lake (pea-toe) is a
            glacier-fed lake located in Banff National Park in the
            Canadian Rockies. The lake itself is easily accessed
            from the Icefields Parkway. It was named for Bill Peyto,
            an early trail guide and trapper in the Banff area.</li>
```



If you would like to see all of the `localStorage` settings, then try downloading and installing the `FireStorage Plus!` plugin from <https://addons.mozilla.org/en-US/firefox/addon/firestorage-plus/>.

All of the content that we cache in this area can now be manipulated using `jQuery` and the `localStorage.getItem` or `localStorage.clearItem` methods. If you would like to learn more, then you may refer to my book *HTML5 Local Storage How-to*, which is available from Packt Publishing.



There is a working version of this code available in the code download that accompanies this book, within the `basicajax-localstorage` folder.

There may be instances where you find you want to reduce the cache TTL value to minutes (or maybe even seconds?). You can do this by modifying lines 70 to 72 in `jquery-ajax-localstorage-cache.js`, and remove one of the multipliers, to leave the following:

```
if ( ! ttl || ttl === 'expired' ) {
    localStorage.setItem( cacheKey + 'cachettl', +new Date() + 1000 *
        60 * hourstl );
}
```

Let's change track . We mentioned earlier that one of the ways we can improve performance when working with AJAX is to ensure that we keep the number of requests to a minimum. If our code contains multiple requests, it will have an adverse impact on performance, particularly if we have to wait for each request to be completed before the next is started.

We could potentially use `localStorage` to reduce the impact, by requesting content from within the browser, instead of the server; it will work, but may not suit every type of request. Instead, as we'll see later, there are better alternatives that allow multiple requests to be handled with ease. Let's delve into this issue in more detail, beginning with the impacts of using callbacks to manage the multiple requests.

Using callbacks to handle multiple AJAX requests

When working with AJAX, we can use the `$.Callbacks` object to manage callback lists – callbacks would be added using the `callbacks.add()` method, fired using `.fire()`, and removed using the `.remove()` method.

Normally we might initiate a single AJAX request if we have decided that content should only appear when needed, and not be present all the time. There is nothing wrong with this – it's a perfectly valid way of working, and reduces the need for page refreshes.

However, if we decided we had to perform multiple requests at the same time, and needed each of them to complete before we could continue, then things will get messy.

```
// Get the HTML, then get the CSS and JavaScript
$.get("/feature/", function(html) {
    $.get("/assets/feature.css", function(css) {
        $.getScript("/assets/feature.js", function() {

            // All is ready now, so...add CSS and HTML to the page
            $("<style />").html(css).appendTo("head");
            $("body").append(html);
        });
    });
});

});
```

We could be waiting for a while!

The problem here is the slow speed of response when working with multiple requests, particularly if all of them have to finish before we can continue. I, for one, certainly don't want to have to wait for a slow responding page to finish!

To avoid what many affectionately term **callback hell**, we can make use of an alternative – jQuery's Deferreds and Promises. These can be thought of as a special form of AJAX. Over the next few pages, we'll dig into what makes this technology tick, and work through a simple example that you can use as a basis for developing your own ideas in the future.



There is even a website dedicated to the horrors of callback hell – you can view it at <http://callbackhell.com/> - it is definitely worth a read!

Let's take a look at how Deferreds and Promises work within jQuery, and how we can use it to enhance our code.

Enhancing your code with jQuery Deferreds and Promises

Although Deferreds and Promises sound like a relatively new technology, they have been available since 1976. In a nutshell:

- A Deferred represents a task that has yet to finish
- A Promise is a value that is not yet known

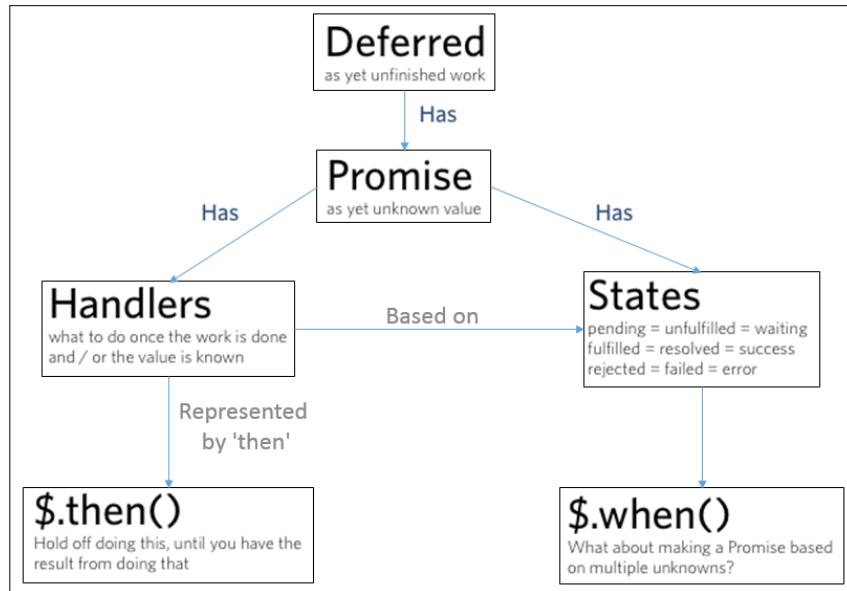
If we have to use standard AJAX, then we will likely have to wait for each request to complete before moving onto the next. This is not necessary with Deferreds / Promises. We do not have to wait for each request to be processed when using Deferreds / Promises. We can queue several to be fired at the same time through the `jQuery.Deferred()` object and manage them individually or together, even though each may take varying amounts of time to complete.

If your application uses, or could benefit from using, AJAX-enabled requests, then it is worth spending the time to familiarize yourself with Deferreds and Promises.

When working with standard AJAX, a key deficiency was the lack of *standard* feedback from any AJAX call – it was difficult to tell when something had been completed. jQuery AJAX now creates and returns a Promise object, that will return a promise when all of the actions bound to it have been completed. Using jQuery, the following way is how we would implement, using the `when()`, `then()` and `fail()` methods:

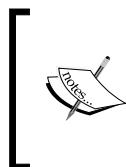
```
$.when($.get("content.txt"))
  .then(function(resp) {
    console.log("third code block, then() call");
    console.log(resp);
  })
  .fail(function(resp) { console.log(resp); });
```

We can represent the principles of working with Deferreds and Promises, using the following diagram:



The key benefit of using Deferreds is that we can begin to chain together multiple functions, instead of being limited to only calling one function at a time (as is the case with standard AJAX). We can then either `.resolve()` or `.reject()` individual Deferreds from within the `jQuery.Deferred` list, and provide a consistent mechanism to determine what should happen if Deferreds are successful or if they fail, using the `.success()`, `.fail()`, or `error()` event handlers.

Finally, we can then call the `.done()` event handler to determine what should happen once the actions bound to our Promise have been completed.



If you would like to learn more about the inner workings of Deferreds and Promises, there is a useful article at <https://github.com/promises-aplus/promises-spec>, although it does make for somewhat dry reading (no pun intended!).



Now that we've covered the basics of Deferreds and Promises, let's change track and take a look at using both in action, along with outlining why it is worth spending time getting acquainted with the concepts behind them.

Working with Deferreds and Promises

Switching to using Deferreds and Promises will take some time, but is worth the effort put into understanding how they work. To get a feel of the benefits of using Deferreds and Promises, let's take a look at some of the advantages of incorporating them into our code:

- **Cleaner method signatures, and uniform return:** We can separate out the code that dictates what happens with the outcome of any request, which makes it cleaner to read and allows chaining if desired, as shown next:

```
$ .ajax(url, settings);
settings.success(data, status, xhr);
settings.error(data, status, errorThrown);
settings.always(xhr, status)
```

- **Easy to put together:** We're not forced to incorporate complex functions to manage handling within each request; this means the core code required to initiate each request is greatly simplified, as shown in the following example:

```
function getEmail(userEmail, onSuccess, onError) {
  $.ajax("/email?" + userEmail, {
    success: onSuccess,
    error: onError
  });
}
```

- **Easy to chain statements together:** The architecture of a Deferred / Promise allows us to chain a number of event handlers together, so that we can fire off a number of methods with a single action, as shown next:

```
$( "#button" ).clickDeferred()
    .then(promptUserforEmail)
    .then(emailValidate)
```

- **Promises always run asynchronously:** They can be fired even when we don't know which callbacks will use the values generated by Promises, before the task completes. Promises will store the resulting value, and we can call that value either from existing callbacks, or any that we add after the promise has been generated.
- **Exception-style error bubbling:** Typically with AJAX, we would have to use a series of if...then...else statements, which makes for a convoluted (and sometimes fragile) way of working. With Promises, we can simply chain together one or more .then() statements to handle any outcome, as shown next:

```
getUser("Alex")
    .then(getFriend, ui.error)
    .then(ui.showFriend, ui.error)
```



There is so much more to Promises than we can cover here. For a useful discussion on comparing Promises with standard AJAX requests, check out this discussion at <http://stackoverflow.com/a/22562045>.

Remember the code we examined back in *Using callbacks to handle multiple AJAX requests*? The key drawback of using multiple callbacks is the resulting mess (and ultimately the impact on performance of our site) – clearly we need a better alternative!

The beauty about Deferreds and Promises is that it allows us to restructure the code to make it easier to read. This includes not only the commands that we need to run as part of the requests, but also what happens if they succeed or fail. Let's revisit that code extract from earlier, and see what it looks like when rewritten to use Deferreds / Promises:

```
$.when(
    // Get the HTML, CSS and JS
    $.get("/feature/", function(html) {
        globalStore.html = html;
    })
),
```

```
$.get("/assets/feature.css", function(css) {
    globalStore.css = css;
}),
$.getScript("/assets/feature.js")
.then(function() {
    // All is ready now, so...add the CSS and HTML to the page
    $("<style />").html(globalStore.css).appendTo("head");
    $("body").append(globalStore.html);
});
```

Hopefully you will agree that it looks significantly cleaner, and that we can now run multiple requests from a single process, without having to wait for each to complete before moving onto the next request!

Time now for some code, I think – let's make use of Deferreds and Promises, and build a demo that uses AJAX. We'll see how we can use it to respond to form submissions, without the need for a page refresh.

Modifying our advance contact form

In the first part of our real-world example, we're going to reuse and develop the basic AJAX form that we created earlier in this chapter, and from the *Developing an advanced file upload form using jQuery* demo in *Chapter 4, Working with Forms*. We will adjust it to display confirmation of submission using AJAX, and for confirmation to also appear as an e-mail.

For this exercise, we will need to avail ourselves of a couple of tools:

- A local web server installed using default settings – options include WAMP (for PC – <http://www.wampserver.de> or <http://www.wampserver.com/en/>), or MAMP (for Mac, <http://www.mamp.info/en/>). Linux users will likely already have something available as part of their distribution. You will need to ensure that your version of PHP is 5.4 or greater, as the code relies on functionality that breaks if an older version is used. You can also try the cross-platform solution XAMPP, available from <https://www.apachefriends.org/index.html> (note that the Test Mail tool is not needed if you use this option – e-mail support is included in XAMPP).

- The free Test Mail Server tool (Windows only), available from <http://www.toolheap.com/test-mail-server-tool/> E-mailing from a local web server can be difficult to set up, so this brilliant tool monitors port 25 and provides local e-mailing capabilities. For Mac, you might try the instructions at <https://discussions.apple.com/docs/DOC-4161>; Linux users can try following the steps outlined at <http://cnedelcu.blogspot.co.uk/2014/01/how-to-set-up-simple-mail-server-debian-linux.html>.
- Access to an e-mail package from the PC or laptop that is being used – this is required to receive the e-mails that are sent using the Test Mail Server tool.

Okay – with the tools in place, let's make a start:

1. We're going to start by opening a copy of the code download that accompanies this book, and extracting the `ajaxform` folder; this contains the markup, styling, and assorted files for our demo. We need to save the folder into the web server's `www` folder, which (for PC) will usually be `C:\wamp\www`.
2. The markup is relatively straightforward, and very similar to what we've already seen throughout this chapter.
3. We need to make one small change to the `mailer.php` file – open it in your text editor of choice, and then look for the following line:

```
$recipient = "<ENTER EMAIL HERE>";
```

Change `<ENTER EMAIL HERE>` to a valid e-mail address that you can use to check that an email has appeared afterwards.

4. The magic for this demo happens within `ajax.js`, so let's take a look at that now, beginning with setting some variables:

```
$(function() {
    var form = $('#ajaxform');
    var formMessages = $('#messages');
```

5. We start the real magic here, when the **Send** button is pressed. We first prevent the form from submitting (as it's default action), then serialize the form data into a string, for submission:

```
$(form).submit(function(e) {
    e.preventDefault();
    var formData = $(form).serialize();
```

6. The core of the AJAX action on this form is next. This function sets the type of request to make, where the content will be sent to, and the data to send:

```
$.ajax({  
    type: 'POST',  
    url: $(form).attr('action'),  
    data: formData  
})
```

7. We then add the two functions to determine what should happen – first one deals with successful submission of our form:

```
.done(function(response) {  
    $(formMessages).removeClass('error');  
    $(formMessages).addClass('success');  
    $(formMessages).text(response);  
    $('#name').val('');  
    $('#email').val('');  
    $('#message').val('');  
})
```

8. Next comes the function that handles the outcome if form submission fails:

```
.fail(function(data) {  
    $(formMessages).removeClass('success');  
    $(formMessages).addClass('error');  
    if (data.responseText !== '') {  
        $(formMessages).text(data.responseText);  
    }  
    else {  
        $(formMessages).text('Oops! An error occurred and your  
        message could not be sent.');    }  
});  
});  
});
```

9. Start the Email Test Server Tool by double-clicking on it. If we preview the form in a browser, and fill out some valid details, we should see the following image when submitting:

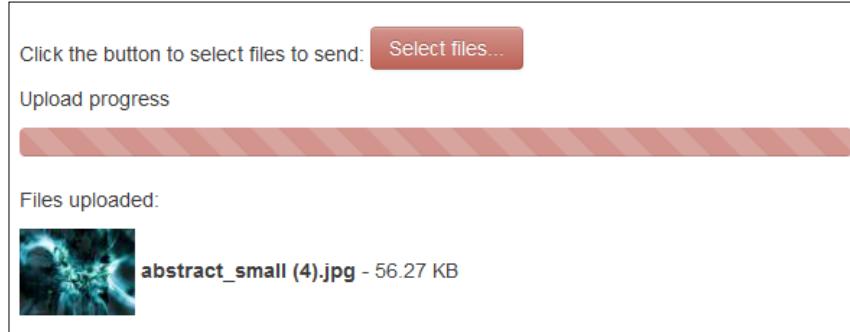
A screenshot of a web application titled "AJAX Contact Form Demo". At the top, there is a red header bar with the title. Below it, a green success message box contains the text "Thank You! Your message has been sent.". Underneath the message box, there is a label "Name:" followed by a red-bordered input field.

Our form is now in place and is able to submit, with confirmation appearing by e-mail within a few moments. We will revisit the use of AJAX within jQuery in greater depth in the next chapter; for now let's move on and continue to develop our form.

Adding file upload capabilities using AJAX

Adding a file upload function is relatively straightforward; it requires both client and server-side components to function correctly.

In our example, we're going to focus more on the client-side functionality. For the purpose of the demo, we will upload files to a fake folder stored within the project area. To give you an idea of what we will build, following is a screenshot of the completed example:



To help us along with this demo, we're going to use the BlueImp file upload plugin; at over 1300 lines long, it's a very comprehensive plugin! This, along with BlueImp's PHP-based file manipulation plugin and some additional jQuery UI, will help with creating a useable file upload facility.



Copies of the plugin files are available in the code download that accompanies this book, or from <https://github.com/blueimp/jQuery-File-Upload>.

Let's make a start:

1. We'll begin by extracting a copy of the `ajaxform-files` folder that is in the code download that accompanies this book – this contains the BlueImp file upload plugins, along with some additional custom CSS and JavaScript files.
2. Add the files from the `ajaxform-files` folder into the `ajaxform` folder that is stored within the `webserver` folder; the JavaScript files should go in the `js` folder, the CSS stylesheet into the `css` folder, and the 2 PHP files can be dropped into the root of our `ajaxform` folder.
3. Next, we need to open a copy of the `ajaxform.html` file from the previous exercise – we first need to add a link to `fileupload.css`, which will contain some additional styles for our upload form:

```
<link rel="stylesheet" href="css/bootstrap.min.css">
<link rel="stylesheet" href="css/styles.css">
<link rel="stylesheet" href="css/fileupload.css">
```

4. We also need to reference the additional JavaScript files that we've just downloaded – add the highlighted links below the reference to `ajax.js` as shown here:

```
<script src="js/ajax.js"></script>
<script src="js/jquery.ui.widget.js"></script>
<script src="js/jquery.iframe-transport.js"></script>
<script src="js/jquery.fileupload.js"></script>
<script src="js/uploadfiles.js"></script>
```

5. Next up comes some markup changes to `index.html`. So in `ajaxform.html`, go ahead and first alter the title as shown next:

```
<div id="formtitle"><h2>AJAX File Upload Demo</h1></div>
<div id="form-messages"></div>
```

6. We now need to add the file upload code, so immediately after closing the `</div>` tag of the message field, add the following code:

```
<div class="container">
    Click the button to select files to send:
    <span class="btn btn-success fileinput-button">
        <span>Select files...</span>
        <input id="fileupload" type="file" name="files[]" multiple>
    </span>
```

```

<p>Upload progress</p>
<div id="progress" class="progress progress-success progress-striped">
    <div class="bar"></div>
</div>
<p>Files uploaded:</p>
<ul id="files"></ul>
</div>

```

7. Save all your files. — If we preview the results using our local web server, then we should expect to see an updated form that now shows a file upload area at the bottom of the form.

 If you would like to see a version with the changes already made, then there is a completed version of this code in the code download that accompanies this book, in the `ajaxform-completed` folder.

Examining the use of Promises and Deferreds in the demo

Although our changes in the second part of this demo are relatively straightforward to make, they hide a wealth of functionality. To get a feel of how AJAX can be used, it is worth looking through the source code of the `jquery.fileupload.js` plugin in detail.

If we open a copy of `ajax.js`, we can see clear use of jQuery's Deferred object, in the form of `.done()`, as shown in the following extract:

```

.done(function(response) {
    $(formMessages).removeClass('error');
    ....
})

```

If however, our AJAX code had failed, jQuery would be executing the methods or functions outlined in the `.fail()` event handler:

```

.fail(function(data) {
    $(formMessages).removeClass('success');
    ....
})

```

If we switch to looking at the code in `uploadfiles.js`, we could be forgiven for thinking that it doesn't use AJAX at all. On the contrary, AJAX is used, but in the `jquery.fileupload.js` plugin.

If we open up the plugin file in a text editor, we can see lots of instances where Deferreds and Promises. Let's take a look at some extracts as examples:

- From the `upload` method - lines 762-766:

```
jqXHR = ((that._trigger('chunksend', null, o) !== false &&
$.ajax(o)) || that._getXHRPromise(false, o.context))
.done(function (result, textStatus, jqXHR) {
```

- In the same method, but this time from lines 794-804:

```
.fail(function (jqXHR, textStatus, errorThrown) {
o.jqXHR = jqXHR;
o.textStatus = textStatus;
```

- This time, from the private `_onSend` method, at lines 900-904:

```
).done(function (result, textStatus, jqXHR) {
that._onDone(result, textStatus, jqXHR, options);
}).fail(function (jqXHR, textStatus, errorThrown) {
that._onFail(jqXHR, textStatus, errorThrown, options);
}).always(function (jqXHRorResult, textStatus, jqXHRorError)
{
```

These are just some examples of how we can use Deferreds and Promises to enhance our code. Hopefully this has given a flavor of what is possible, and how we can dramatically improve not only the readability of our code, but also the resulting performance of our projects.

Detailing AJAX best practices

Throughout this chapter, we've revisited the basics, and explored some of the techniques we can use to take our knowledge of AJAX to the next level - the key being that it is not necessarily just about coding, but visiting some of those tips and tricks that help make us a more rounded developer.

In *Working with Deferreds and Promises* section, we explored the basics of using jQuery's Deferreds and Promises, and how the change in architecture when using them can lead to significant improvements in performance. Before we round up this chapter, there are some additional best practices that we should follow wherever possible. Following list explains them:

1. There is no need to call `.getJSON()` or `.get()` directly. These are called when using the `$.ajax()` object by default.

2. Don't mix protocols when calling requests. The preference is to use schemaless requests where possible.
3. If you are just making GET requests, try to avoid putting request parameters in the URL – instead send them using the `data` object setting, thus:

```
// Less readable
$.ajax({
  url: "something.php?param1=test1&param2=test2",
  ...
});

// More readable
$.ajax({
  url: "something.php",
  data: { param1: test1, param2: test2 }
});
```

4. Try to specify the `dataType` setting so it's easier to know what kind of data you are working with. For an example, please refer to the *Creating a simple example using AJAX*, from earlier section in the chapter.
5. Use delegated event handlers for attaching events to content loaded using AJAX. Delegated events can process events from descendant elements that are added to the document at a later time:

```
$("#parent-container").on("click", "a", delegatedClickHandler);
```



To learn more, please refer to <http://api.jquery.com/on/#direct-and-delegated-events>.

Summary

AJAX as a technology has been around for years. It can arguably be seen as a game-changer, where the use of JavaScript killed the need to continually refresh page content within a browser. jQuery has helped to enhance this group of technologies. In this chapter we revisited some of the basics, before exploring how best to take our development skills further. Let's recap what we've learnt:

We kicked off with a brief look back at what AJAX is, and reminded ourselves of the basics of constructing an AJAX request within jQuery.

Next, we took a look at some of the tips and tricks we can use to improve the speed of loading from static sites; we picked up one additional trick in the form of using `localStorage` to cache content. We then moved onto discussing how implementing callbacks can make code messy and slow, before moving onto seeing how Deferreds and Promises can improve our code, and ultimately the performance of our sites.

We finished off with a look at a demo, where we borrowed one of the forms from *Chapter 4, Working with Forms*, and extended it by first adding an AJAX based-notification, then by making use of the BlueImp plugin to incorporate a file upload facility, that made use of Deferreds and Promises.

In the next chapter, we're going to expand on one of my personal favorites. It's time to get animated, as we take a look at using jQuery to bring life to elements on our websites.

6

Animating in jQuery

Hands up who like a static website? Thought not, animating a website gives it life; overdoing it can be disastrous!

Two common effects we frequently use to help breathe life into any website are AJAX and animation; we covered the former in detail back in the previous chapter. In this chapter, we'll take a look at when to use jQuery over CSS (or vice versa), how to manage queues better, and how to implement some slick custom animation effects. You'll also see how you can easily create some useful custom-easing effects, as a basis for converting them to CSS equivalents at some point in the future. In this chapter, we'll cover the following topics:

- When to use CSS over jQuery
- Managing or avoiding the jQuery animation queue
- Designing custom animations
- Implementing some custom animations
- Animating in a responsive website

Ready to make a start? Let's go...

Choosing CSS or jQuery

Let's start this topic with a question.

Take a look at the Dia do Baralho site, hosted at <http://www.diadobaralho.com.br> - how many of you think the animations you see there, were created using just jQuery?

If you thought yes, then sorry to disappoint you; the answer is actually no! If you look closely at the source, you will find instances where a mix of both CSS3 animations and jQuery have been used. Now, you might be thinking: why are we talking about CSS3 animations when this book is about mastering jQuery?

There's a good reason for this; remember when I mentioned, earlier in the book, that any individual with the right skills can write jQuery? The difference between an average coder and a good developer is this: why will I use jQuery? Now, this might sound as though I've really lost my marbles, but I haven't. Let me explain what I mean, as follows:

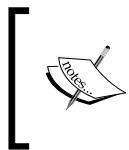
- A CSS3 animation does not have any dependency on an external library; given that jQuery still weighs in at a good size, one less resource request is always a good thing!
- For light, simple animations, there is no benefit in referencing jQuery when CSS3 animations are sufficient. Despite the need to provide vendor-prefixed versions of the same statements (and excluding the use of jQuery), the amount of code required is likely to be smaller than that required if jQuery is used.

There is a performance impact in using jQuery, which makes using CSS animations all the more tempting, for several reasons:

- The library was never designed to be a performant animation engine; it's code base has to serve many purposes, which can lead to layout thrashing
- jQuery's memory consumption often means that we have garbage collections take place, which can lead to the momentary freezing of animations
- jQuery uses `setInterval` instead of `requestAnimationFrame` to manage animations (although this is due to a change in a forthcoming version of jQuery)

There are an equal number of reasons why we should prefer to use jQuery; even though it has its limitations as a library, there are occasions where we may need to use jQuery in place of native CSS3 animations, as stated here:

- CSS animations are taxing on GPUs, which can result in stuttering and banding when the browser is under load – this is particularly prevalent in mobile devices.



There is a useful discussion about the impact of hardware acceleration and CSS3 at <http://css-tricks.com/myth-busting-css-animations-vs-javascript/>.



- Most browsers support CSS3 animations with the exception of IE9 or below; for this, jQuery must be used.
- CSS3 animations are not (yet) as flexible as their jQuery equivalents – they are evolving all the time, so there will come a point when the two become very similar. For example, we cannot use different eases in keyframes when working with CSS3; the same ease must be applied to the whole keyframe.

The key point here is that we have the freedom to choose; in fact, as noted by the developer David Walsh, it is more sensible to use CSS3 animations when we need nothing more than simple state changes. His argument is based on being able to retain animation logic within style sheets and reducing bloat on pages from multiple JavaScript libraries.

The proviso though is that if your needs are more complex, then jQuery is the way forward; developer Julian Shapiro argues that using animation libraries maintains the performance of each animation and keeps our workflow manageable.



To see the effects of animating multiple objects using JavaScript or CSS, head over to <http://css3.bradshawenterprises.com/blog/jquery-vs-css3-transitions/>, which shows a very enlightening demo!



As long as we are careful with our CSS, for simple, self-contained state animations, a smarter move is to use native CSS3 and to not always rely on using jQuery as the answer to all our needs.

As an aside, it is worth noting that a relatively new API is being considered: the Web Animations API. This API is aimed at creating animations using JavaScript, that run as efficiently as native CSS3 animations. This is worth looking out for, given the inherent issues we have with using jQuery; support is limited to Chrome and Opera only at the time of writing.

[ For details of the support of the Web Animations API, check out the Can I use website at <http://caniuse.com/#search=Web%20animation>; there is also a useful tutorial posted at <http://updates.html5rocks.com/2014/05/Web-Animations---element-animate-is-now-in-Chrome-36>—this is for Chrome only though!]

Enough of theory, let's do some coding! Assuming that we need to use jQuery for our animation projects, there is one key issue that is likely to floor developers: rapid cycling through queued animations that can be set for any feature that uses animation. Let's delve in to see what this means and what we can do to reduce or get rid of the issue.

Controlling the jQuery animation queue

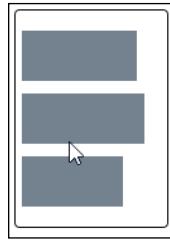
If you have spent any time developing with jQuery, there is no doubt that you will come across a key issue when working with animations: how many times have you seen a browser cycle through multiple queued animations when you switch to another browser window and back again?

I can bet that the answer is quite a few times; the key to this issue boils down to jQuery queuing all the animations it has been asked to perform. If too many initiations take place, then jQuery's animation queue becomes confused and hence it seems to go crazy! Let's take a look at the issue in action before working through a simple fix for the problem:

1. Start by extracting the `blockedqueue.html` and `blockedqueue.css` files from the code download that accompanies this book—they will provide some simple markup to illustrate our queuing issue.
2. In a text editor, add the following to a new file, saving it as `blockedqueue.js` in the `js` subfolder of our project area:

```
$(document).ready(function() {  
    $(".nostop li").hover(  
        function () { $(this).animate({width:"100px"},500); },  
        function () { $(this).animate({width:"80px"},500); }  
    );  
});
```

3. If we run our demo now, then when we repeatedly move the mouse over each bar, we can see all of them increase or decrease in quick succession, with the next bar changing before the previous one has finished animating, as shown here:



Clearly, this behavior isn't desired; had this demo been automated and set to work in conjunction with `requestAnimationFrame` (which we will cover later in *Chapter 9, Using the Web Performance APIs*), then we would have seen a frenzied rush of animations being completed when we switch away from a tab and go back to the original.

Fixing the problem

How do we fix this issue? It's really simple; all we need to do is add the `.stop()` method in our statement chain; this will clear the preceding animation before starting the next. Let's take a look and see what this means in practice by performing the following steps:

1. In a copy of the `blockedqueue.html` file, go ahead and modify the `<head>` section as shown here:

```
<title>Demo: Clearing the animation queue</title>
<link rel="stylesheet" href="css/blockedqueue.css">
<script src="js/jquery.min.js"></script>
<script src="js/unblockqueue.js"></script>
</head>
```

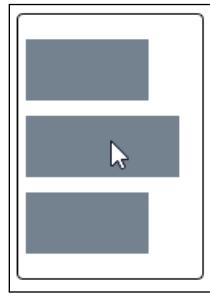
2. We need to slightly change the markup in the body of our demo, so alter the code as highlighted:

```
<div id="container">
  <ul class="stop">
    <li></li>
```

3. Save this as `unblockqueue.html`. In a new file, add the following code and then save it as `unblockqueue.js` in the `js` subfolder of our project area. This contains the modified markup, with the addition of `.stop()`:

```
$(document).ready(function() {
    $(".stop li").hover(
        function () {
            $(this).stop().animate({width:"100px"},500);
        },
        function () {
            $(this).stop().animate({width:"80px"},500);
        }
    );
});
```

4. If we run the demo now and then rapidly move over each of the bars in turn, we should see that the bars will increase and decrease in turn, but the next one will not change until the preceding bar has returned to its original size, as shown here:



Hopefully, you will agree that adding `.stop()` has made a significant improvement to our code—adding `.stop()` will terminate the previous animation but queue the next one, ready for action.

Making the transition even smoother

We can go one step further. A closer look at the attributes available for `.stop()` shows that we can use `clearQueue` and `jumpToEnd` to stop running animations on matched elements, making for even cleaner transitions, as shown in the following figure:

The jQuery .stop() method explained		
<code>\$(".test").stop("string", boolean, boolean).animate(...);</code>		
queue	clearQueue	jumpToEnd
The name of the queue in which to stop animations	A boolean indicating whether to remove queued information - defaults to <code>false</code> .	A boolean indicating whether to complete the current animation immediately - defaults to <code>false</code> .



For more information about using `.stop()`, please refer to the main jQuery documentation at <http://api.jquery.com/stop/>.

Let's alter our jQuery code to see what this means in practice by performing the following steps:

1. Go back to the `unblockedqueue.js` file and then alter the code as shown here:

```
function () {
    $(this).stop(true, false).animate({width:"100px"},500);
},
function () {
    $(this).stop(true, false).animate({width:"80px"},500);
}
```

2. Save your work and then preview the results of the demo in a browser. If all went well, you should see no change in the bars themselves but the animation effect will appear smoother when you hover over each bar.

At this stage, we should have an animation that still works but with a smoother transition – it is worth noting that this trick will only work with animations. If your projects use other function queues, then these will need to be cleared using `.clearQueue()` instead.



For a comparison on the different ways of using `.stop()`, it's worth taking a look at a demo by Chris Coyier, at <http://css-tricks.com/examples/jqueryStop/> – this produces some intriguing effects! A similar explanation is also available at <http://www.2meter3.de/code/hoverFlow/>.

Using a pure CSS solution

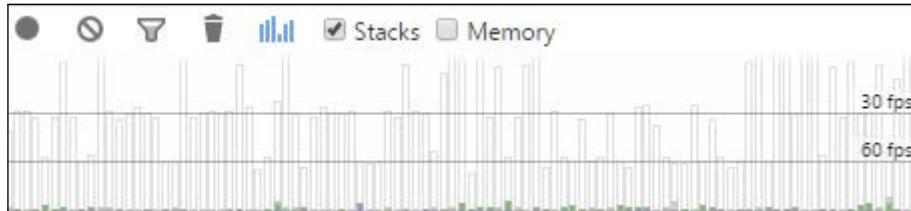
Okay, so we have our animation in jQuery; for a simple animation, what will it look like if we used pure CSS instead? Although we can't replicate the same effect as `.stop()`, we can get pretty close. Let's take a look and see what this means in practice, using `unblockqueue.html` as the basis for our demo:

1. Start by removing the two JavaScript links, one to `unblockqueue.js` and the other to jQuery itself.
2. Add the following at the bottom of `blockqueue.css`—this contains the animation style rules required for our demo:

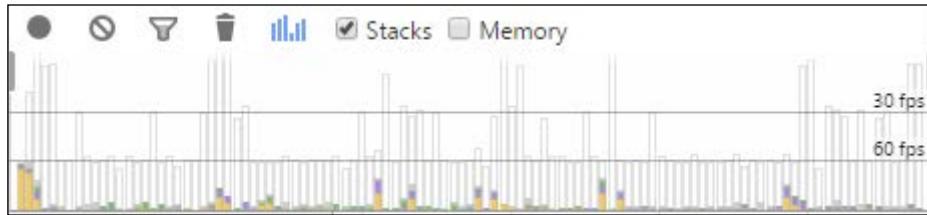
```
li { width: 50%; transition: width 1s ease-in, padding-left 1s ease-in, padding-right 1s ease-in; }
li:hover { width: 100%; transition: width 1s ease-out, padding-left 1s ease-out, padding-right 1s ease-out; }
```

At this point, if we preview the results in a browser, we should see no *visible* difference in our animated list elements; the real change is seen if we use Google Chrome's developer toolbar to monitor the timeline. Let's see what the change looks like.

1. Fire up Google Chrome. Press `Shift + Ctrl + I` to bring up the **Developer Toolbar** (or `Option + Cmd + I` for Apple Macs).
2. Click on the **Timeline** tab and then click on the gray-colored circle immediately below the magnifying glass—the circle will turn red in color.
3. Try hovering over the list items in turn; Chrome will monitor and collect details of the actions performed.
4. After a couple of minutes, click on the red-colored circle to stop generating the profile; you will end up with something that looks like this:



We can clearly see that a CSS-only solution barely has an impact on the performance of the browser. In comparison, take a look at the same timeline, when we run the `unblockedqueue.html` demo:



Notice the difference? Although this was a quick, nonscientific test, we can clearly see the difference when we look at the detailed numbers.

Over a period of approximately 3 seconds, Google Chrome spent 33 ms rendering and 48 ms painting when running the CSS-only solution. Running `unblockedqueue.html` shows that the numbers almost double: 107 ms for scripting, 78 ms for rendering, and 76 ms for painting! This is definitely something to think about...

Improving jQuery animations

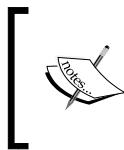
From the previous section, we can easily see that CSS has a clear advantage when being rendered in a browser – this is despite the somewhat unscientific approach used in the demo!

The key point though is that, what we gain with flexibility and all-round browser support when using jQuery, we lose in speed – jQuery was never designed to be performant when rendering animations.

To help improve performance, there are a couple of plugin options that you can explore:

- **Velocity.js**: This plugin reengineers `$.animate()` to provide significantly faster performance and can be used with or without jQuery; this includes IE8. The plugin can be downloaded from <http://julian.com/research/velocity/>. This also contains some preregistered effects – we will cover more on creating custom-easing effects later in this chapter.
- **jQuery-animate-enhanced**: This plugin detects and reengineers animations to use native CSS transitions automatically, for WebKit, Mozilla, and IE10 or greater. It can be downloaded from <http://playground.benbarnett.net/jquery-animate-enhanced/>.

We can go further and delve into using jQuery to be notified when an animation has completed, using the `transitionend` event. While this may not stop the original issue with an animation queue build-up, using jQuery will allow you to separate animation effects from your jQuery logic.



For an interesting article and demo on using `transitionend` (and its vendor-prefixed versions), take a look at an article on the Treehouse website, at <http://blog.teamtreehouse.com/using-jquery-to-detect-when-css3-animations-and-transitions-end>.

Now that we've seen how we can make our animations smoother, let's move on and take a look at how we can generate custom animations; the theory being that we can put some of our knowledge to create more complex and interesting animations, while at the same time, reduce some of the issues we see with running the queue.

However, before we do so, I want to leave you with two useful tips when it comes to improving your animations:

- Have a look at <http://blog.teamtreehouse.com/create-smoother-animations-transitions-browser>; it explores some of the issues we encounter with animations and transitions and how these affect performance
- The article at <http://developer.telerik.com/featured/trimming-jquery-grunt/> explores how we can trim our version of jQuery, to remove functionality that is not needed (and consequently reduce the load on the server when running animations)

Let's take a look at designing these custom animations, beginning with an initial look at using easing functions.

Introducing easing functions

When animating any object or element on a page, we can simply slide it up or down or move it from one place to another on the page. These are perfectly valid effects, but they lack the realism you might get when opening a drawer, for example.

Animations don't always move at a constant speed; instead, we might get a little bounce back if we were bouncing a ball or a slow down when opening a chest of drawers. To achieve this effect, we need to use easing functions, which control the rate of change. There are plenty of examples available on the Internet – a great place to start is <http://www.easings.net> – or perhaps we can watch the effects on sites such as <http://matthewlein.com/ceaser/>. Over the next few pages, we're going to explore these in more detail and look at tips and tricks that we can use to push our animation skills to a new level.

Designing custom animations

If you've spent any time developing jQuery code that animates objects or elements on a page, you will no doubt have used either the jQuery UI or possibly a plugin, such as jQuery Easing, created by George Smith (<http://gsgd.co.uk/sandbox/jquery/easing/>).

Both are great ways of animating objects on a page, using easing methods such as `easeIn()` or `easeOutShine()`. The trouble is that both require the use of plugins, which add unnecessary baggage to our code; they are also a very safe way of achieving the effect we need. What if I said we don't need either and can produce the same effects just by using jQuery itself?

Before I go through how to do this, let's take a look at a working demo that shows this in action:

1. Let's make a start by extracting the relevant files from the code download that accompanies this book—for this demo, we will need copies of the following:
 - `customanimate.html`: Save this file in the root area of our project folder
 - `customanimate.css`: Save this file in the `css` subfolder of our project folder
 - `customanimate.js`: Save this file in the `js` subfolder of our project folder

Open the Sans font; save this in the `font` folder of our project folder; alternatively, the font is available at <http://www.fontsquirrel.com/fonts/open-sans>.

2. If you preview the `customanimate.html` file in a browser and then run the demo, you should see something akin to this screenshot, where the `<div>` tag is partway through running the animation:



So, what happened here? Well, we've used nothing more earth-shattering than a standard `.animate()` to increase the size of and move the `<div>` tag to its new location.

There's nothing new here then, right? Wrong, the "new" bit here is actually how we constructed the easing! If you take a look at `customanimate.js`, you will find this code:

```
$ (document) .ready(function() {
  $.extend(jQuery.easing, {
    easeInBackCustom: function(x,t,b,c,d) {
      var s;
      if (s == undefined) s = 2.70158;
      return c* (t/=d)*t*((s+1)*t - s) + b;
    }
  })
})
```

All we've done is take the math needed to achieve the same effect and wrapped it in a jQuery object that extends `$.easing`. We can then reference the new easing method within our code, as follows:

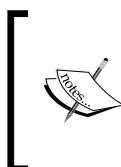
```
$ ("#go") .click(function() {
  $("#block") .animate({
    ...
  }, 1500, 'easeInBackCustom');
});
```

This opens up lots of possibilities; we can then replace the custom-easing function with our own creation. A trawl of the Internet threw up lots of possibilities, such as these two examples:

```
$.easing.easeOutBack = function(t) {
  return 1 - (1 - t) * (1 - t) * (1 - 3*t);
};

$.easing.speedInOut = function(x, t, b, c, d) {
  return (sinh((x - 0.5) * 5) + sinh(-(x - 0.5)) + (sinh(2.5) +
  Math.sin(-2.5))) / (sinh(2.5) * 1.82);
};
```

To really get stuck into understanding how easing functions work is outside the scope of this book—if you are interested in the math behind it, then there are several sites on the Internet that explain this in greater detail.



Two examples of how to work with easing functions include <http://upshots.org/actionscript/jsas-understanding-easing> and <http://www.brianwalg.com/journal/creating-custom-jquery-easing-animations>—note that they do make for dry reading though!

Suffice to say that the best source for the easing functions is the source code for jQuery, where we can view each of the calculations required and use these as a basis for creating our own easing effects.

This is all well and good; it's a great way to achieve good animations without producing complex code that is difficult to understand or debug. But...you know me; I think we can still do better. How? That's easy, what if we can replicate some of the easing effects we might see in CSS transitions in jQuery?

Converting to use with jQuery

At this point, you probably think I really have lost it now; CSS transitions use Bezier curves, which are not supported when working with jQuery's `animate()` method. So, how can we achieve the same effect?

The answer lies, as always, with a plugin—granted, this goes against what we've talked about in the previous demo though! However, there is a difference: this plugin weighs in at 0.8 KB when compressed; this is significantly smaller than using the jQuery UI or the Easing plugin.

The plugin that we're going to use is the Bez plugin by Robert Grey, available at <https://github.com/rdallasgray/bez>; this will allow us to use cubic-bezier values, such as `0.23, 1, 0.32, 1`, which is the equivalent of `easeOutQuint`. Let's take a look at this in action:

1. We first need to download and install the Bez plugin—we can download it from GitHub at <https://github.com/rdallasgray/bez>; add a reference to it from within `customanimate.html`, immediately underneath the link to jQuery.
2. Next, open up a copy of `customanimate.js`; go ahead and alter this line as shown, which replaces the `easeInBackCustom` action we used earlier:

```
}, 1500, $.bez([0.23, 1, 0.32, 1]));
```

Save both the files; if you preview the results in a browser, you will see a different action when running the demo as compared to what you saw in the previous example.

So, how did we get here? The trick behind this is a combination of the plugin and the easings.net website. Using `easeOutQuint` as our example easing, if we first visit <http://easings.net/#easeOutQuint>, we can see the cubic-bezier values required to produce our effect: `0.86, 0, 0.07, 1`. All we need to do is insert this into a call to the Bez plugin and we are all set:

```
}, 1500, $.bez([0.86, 0, 0.07, 1]));
```

If, however, we want to create our own cubic-bezier effect, then we can use cubic-bezier.com to create our effect; this will give us the values we need to use, as shown in the following screenshot:

cubic-bezier(.17,.67,.83,.67)

We can then plug these into our object call in exactly the same way as we did in the previous example. The beauty of using this method is that we have an easy route to convert the animations to CSS3 equivalents, should we later decide to reduce our usage of jQuery at some point in the future.



To learn more about the theory behind Bezier curves, take a look at the Wikipedia article available at http://en.wikipedia.org/wiki/B%C3%A9zier_curve.



Okay, so we've covered how to create our own animation-easing functions; what if we wanted to use effects available from existing libraries? No problem, there are some good examples available on the Internet, which include the following:

- <http://daneden.github.io/animate.css/>: This is the home of the Animate.css library; we can reproduce the effects within this library using the `jQuery.Keyframes` plugin available at <https://github.com/jQueryKeyframes/jQuery.Keyframes>.
- <https://github.com/yckart/jquery-custom-animations>: This library contains a number of different effects, created in a style similar to the jQuery UI; this can be dropped in and the effects can be referenced in a similar fashion to the *Designing custom animations* demo from earlier in this chapter.
- <https://github.com/ThrivingKings/animo.js>: Animo.JS takes a different approach; instead of using jQuery's `animate()` function, it uses its own `animo()` method to animate objects. It uses the effects from the Animate.css library, created by Dan Eden – although one might argue whether it is worth the extra overhead, it is nonetheless worth a look as a possible source of animations for your projects.
- <http://lvivski.com/anim/:> It's worth taking a look at this library carefully; the source code contains a number of cubic-bezier values within the `easings.js` source file. These can be easily lifted into your own code projects if desired or can provide inspiration for your own examples, perhaps.

It's time to put some of the animation concepts we've covered to good use; let's move on and take a look at some of the examples of using animation in our own projects.

Implementing some custom animations

Throughout this chapter, we've explored the use of jQuery to animate objects and seen how this can be compared with CSS-based animations; we've also looked at creating some custom-easing patterns that control how the elements are moved on screen.

Enough of the theory, let's get stuck into some practical uses! Over the next few pages, we're going to take a look at some examples of animating elements; we will include some examples for responsive sites, as this is a popular topic, with the rise in the use of mobile devices to access content on the Internet.

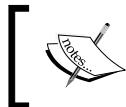
Let's make a start, with a look at animating a simple element, in the form of some buttons – watch out for the twist at the end of the demo!

Animating rollover buttons

The humble button must be one of the most important elements on any website; buttons come in all shapes and sizes, and can be created from the standard `<button>` HTML element, or by the use of an `<input>` field.

In this demo, we're going to use jQuery to not only slide in and slide out the button icons, but also to rotate them at the same time. But hold on – we all know that jQuery doesn't support the rotating of elements, right?

We could can use plugins such as QTransform (<https://github.com/puppybits/QTransform>) or even jQuery Animate Enhanced (<http://playground.benbarnett.net/jquery-animate-enhanced/>), but this has an overhead – let's take a different route. Instead, we'll use a monkey patch to directly retrofit support; to prove that it works, we'll update a **Codrops** demo, which had the original version of the rolling buttons on its site, to use jQuery 2.1 instead.



The original version of this demo is available at <http://tympanus.net/codrops/2010/04/30/rocking-and-rolling-rounded-menu-with-jquery/>.



Let's take a look at the demo:

1. Extract the relevant files from the code download that accompanies this book; for this demo, we will need the following files:
 - `rollingbuttons.html`: Save this file in the root subfolder of your project area
 - `style.css`: Save this file in the `css` subfolder of your project area
 - `jquery-animate-css-rotate-scale.js`: Save this file in the `js` subfolder of your project area
 - `rollingbuttons.js`: Save this file in the `js` subfolder of your project area
 - `img`: Copy this folder to the project area

[ The original version of this monkey patch is available at <http://www.zachstronaut.com/posts/2009/08/07/jquery-animate-css-rotate-scale.html>; it was developed for jQuery 1.3.1+, but I have not seen any adverse effects when I used it with jQuery 2.1.]

2. Run the demo in a browser and then try hovering over one or more buttons. If all is working OK, then we will see the green-colored icon image start to spin out to the left while the gray background expands to form a long pill, with the links held within, as shown here:



Exploring the code in more detail

This demo produces a nifty effect as well as acts as a space saver; the information is only exposed when visitors need to view it and is hidden at all other times.

If we delve into the code in more detail though, it reveals an interesting concept: jQuery does not offer native support for the use of rotating elements when using `.animate()`, as mentioned at the start of this demo.

So, how are we going to get around this? We can use a plugin, but instead, we're using a monkey patch (created by the developer Zachary Johnson) to retrofit support to jQuery. It's worth noting that there is always a risk in using patches (as outlined in *Chapter 2, Customizing jQuery*), but in this instance, it seems that despite the update to using jQuery 2.1, there are no noticeable ill-effects.

If you want to see the difference when the patch is being used, activate a DOM Inspector, such as Firebug, before running the demo. Hover over one of the icons; you should see something akin to this screenshot:

```
element.style {  
    transform: matrix(0.766045, -0.642787, 0.642787,  
    0.766045, 0, 0) rotate(40deg);  
}
```

If you want more in-depth details on how `matrix()` works, then visit the notes on Mozilla's site, at <https://developer.mozilla.org/en-US/docs/Web/CSS/transform>.

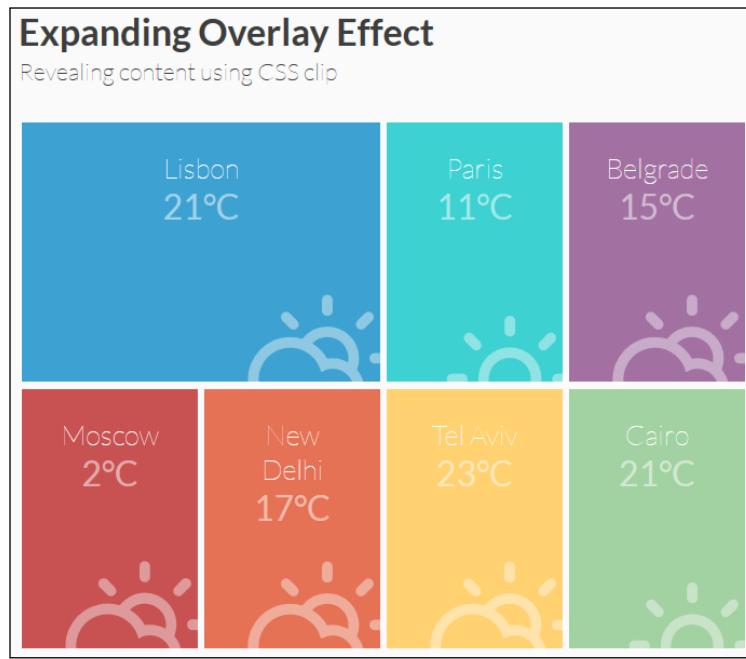
Let's move on and take a look at our next animation example. I'm sure you've used an overlay in some form or other, but we're going to take a look at an overlay that takes a whole new approach and does away with the gray mask that is typical with most overlays.

Animating an overlay effect

If you've spent any time visiting websites on the Internet, you will no doubt have come across ones that use some form of overlay, right?

You know the drill: they first black out the screen with a semitransparent overlay and then begin to display an enlarged version of an image or video. It's a typical effect found on thousands of sites worldwide and can be very effective if put to good use.

However, you know me better than this; I like to take things further! What if we break the tradition and have an overlay that doesn't show an image but shows an all-over display? Intrigued? Let's take a look at what I mean in action:



For this demo, we're going to run a version of the overlay effect, shown at <http://tympanus.net/Tutorials/ExpandingOverlayEffect/>.

1. Let's start by extracting the following files from the code download that accompanies this book; save them in the relevant folders within your project area:
 - `jquery.min.js`: Save this file in the `js` subfolder of your project area
 - `fittext.js`: Save this file in the `js` subfolder of your project area
 - `boxgrid.js`: Save this file in the `js` subfolder of your project area
 - `default.css`, `component.css`, and `climacons.css`: Save these files in the `css` subfolder of your project area
 - `overlayeffect.html`: Save this file in the root of your project area
2. Run `overlayeffect.html` and then try clicking on one of the colored boxes.

Notice what happens? It displays an overlay effect as you expected, but this one covers the entire browser window and does not have the mask effect that is frequently displayed with the more traditional overlay effects.

In this demo, we've used a mix of HTML to produce our initial grid; the `fittext.js` plugin is used to help ensure that the text (and consequently the overlay) is resized to fill the screen; the overlay effect is produced using the `boxgrid.js` plugin from within our code.

The magic happens in `boxgrid.js`—this contains the `jquery.debouncedresize.js` plugin by Louis Remi; although this is 2 years old; it still works perfectly well within modern browsers. You can download the original plugin from <https://github.com/louisremi/jquery-smartresize/blob/master/jquery.debouncedresize.js>.

Let's change focus and move on to take a look at how you can equally apply jQuery animation to responsive websites. In the first of two demos, you'll see how to apply a mix of CSS3, jQuery, and `history.pushState` in order to create some pleasing transition effects that can turn a multiple-page site into what appears to be a single page application.

Animating in a responsive website

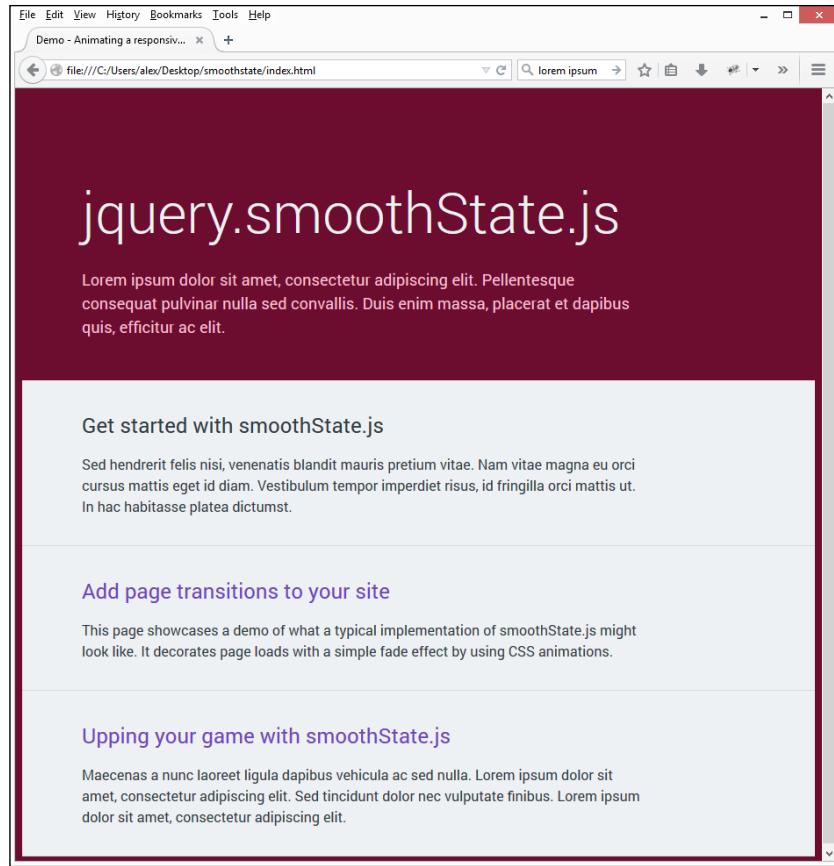
How often have you visited a site only to find out that you have to wait for ages between each page load? Sounds familiar?

Our expectations of page transitions have changed over the last few years—the chunky side effects of elements rearranging on a page will not suffice; we expect more from a website. JavaScript-based **Single Page Application (SPA)** frameworks are often seen as the answer, but at the expense of having to use obtrusive code.

We can do much better than this. We can introduce `smoothState.js`, a useful plugin created by Miguel Ángel Pérez, that allows us to add transitions to make the whole experience smoother and more enjoyable for visitors. In this example, we're going to use a modified version of the demo provided by the plugin's author; some of the code has been reorganized and cleaned up from the original.

Let's take a look at the plugin in action and see how it can make for a much smoother experience. To do this, perform the following steps:

1. From the code download that accompanies this book, extract copies of the following files:
 - `smoothstate.html` and `smoothstate.css`: Save these files in the root area and `css` subfolder of your project folder, respectively.
 - `jquery.smoothstate.js`: Save this in the `js` subfolder of your project area; the latest version can be downloaded from <https://github.com/miguel-perez/jquery.smoothState.js>.
 - `jquery.min.js`: Save this in the `js` subfolder of your project area.
 - `animate.css`: Save this in the `css` subfolder of your project area; the latest version is available at <http://daneden.github.io/animate.css/>.
 - The Roboto font: A copy of the two fonts used are in the code download that accompanies this book. Alternatively, they can be downloaded from the Font Squirrel website, at <http://www.fontsquirrel.com/fonts/roboto>. We only need to select the WOFF font; we will use the light and regular versions of the font in our demo.
2. Run the `smoothstate.html` file in a browser; try clicking on the middle link of the three and see what happens. Notice how it displays the next page, which is `transitions.html`. Instead of the pause we frequently get when loading new pages, `smoothState.js` treats the site as if it were a SPA, or single page application. You should see a very simple page display, as shown in this screenshot:



Traditionally, when faced with this issue, many might resort to an SPA framework in order to fix the issue and improve the transition appearance. Using this approach will work, but at the expense of the benefits gained from using unobtrusive code.

Instead, we can use a mix of jQuery, CSS3, `history.pushState()`, and progressive enhancement to achieve the same effect, resulting in a better experience for our end users.

[ It's worth taking a look at the website documentation, available at <http://weblinc.github.io/jquery.smoothState.js/index.html>. There is a useful tutorial on the CSS-Tricks website, at <https://css-tricks.com/add-page-transitions-css-smoothstate-js/>.]

Maintaining a good user experience should always be at the forefront of any developer's mind – this is more important when working with responsive sites. A key part of this should be to monitor the performance of our animations, in order to ensure that we get a good balance of user experience against the demand on our servers.

There are a few tricks that we can use to help with performance when it comes to using jQuery-based animations on responsive sites. Let's take a look at some of the issues and how we can either mitigate or resolve them.

Considering animation performance on responsive sites

In this modern age of accessing the Internet from any device, the emphasis on user experience is more critical than ever – this is not helped when jQuery is used. Acting as the lowest common denominator, it helps to simplify working with content (particularly for complex animations) but is not optimized for their use.

There are a few issues that we will come across when animating content using jQuery – we've covered some of them earlier in the chapter, in *Choosing CSS or jQuery*; they apply equally to responsive sites. In addition, there are other considerations we need to be aware of, which include the following:

- Animations that use jQuery will consume a lot of resources; this coupled with content that might not suit a mobile environment (due to its volume) will create a slow experience on desktops. This will be even worse on laptops and mobile devices!
- End users on a mobile device are frequently only interested in getting the information they need; animations may make a site look good but are often not optimized for mobile devices and are likely to slow access and result in the browser crashing.
- jQuery's garbage collection process is frequently known to cause issues; its use of `setInterval()` in place of `requestAnimationFrame()` will result in high frame rates, making for an experience that is likely to stutter and show a high rate of frame dropouts.



At the time of writing, there are plans to replace `setInterval` (and `clearInterval`) in jQuery with `requestAnimationFrame` (with `clearAnimationFrame`).

- If we are using animations—both jQuery or plain CSS—then on some platforms, we frequently need to enable hardware acceleration. While this can help with performance on mobile devices, it can also lead to flickering if hardware-accelerated elements overlap with other elements that are not hardware-accelerated. We will touch on how to enable 3D rendering later in this chapter, in the *Improving the appearance of animations* section.
- jQuery's `.animate` increments the element's `style` attribute on every animation frame; this forces the browser to recalculate the layout and leads to continual refreshes. This is particularly acute on responsive sites, where each element needs to be redrawn each time the screen is resized; this will place additional demands on server resources and impact performance. If desired, plugins such as jQuery Timer Tools (<https://github.com/lolmaus/jquery.timer-tools>) can be used to throttle back or delay actions so that they are only executed when necessary, or multiple repetitive calls are effectively merged into one single execution.
- If the display state of elements is changed (using `display...` or `display: none`), then this has the effect of adding or removing elements from the DOM. This can have an impact on performance, if your DOM is heavy with lots of elements.
- Using jQuery leaves inline styles in the DOM that have very high specificity and that will override our well-maintained CSS. This is a big issue if the viewport is resized and triggers different breakpoints.

 CSS specificity is where the browser decides which property values are most relevant to the elements and are applied as a result—check out <https://css-tricks.com/specifics-on-css-specificity/> for more details.

- As an aside, we lose the separation of concerns (or defining separate sections for our code) as styles are defined in JavaScript files.

Is it possible to reduce or remove these issues? Yes, but it's likely to require some sacrifices; these will depend on what your requirements are and the target devices that need to be supported. Let's take a moment to consider where we can make changes:

- Consider the use of CSS over jQuery where practical, at least for mobile sites; most browsers (with the exception of Opera Mini) support CSS keywords such as `translate` or `transform`. As they are native to the browser, this removes the reliance on the extra code being referenced, resulting in the resources and bandwidth usage being saved.

- If animation isn't possible using jQuery or the effort required outweighs the benefits gained, then consider the use of a plugin such as Velocity.js (available from <https://github.com/juliansapiro/velocity>), as this has been optimized to animate content.

 It's worth noting that discussions are being held to integrate Velocity.js into jQuery – for more details, see <https://github.com/jquery/jquery/issues/2053>. There is also a post that is worth reading at <http://www.smashingmagazine.com/2014/09/04/animating-without-jquery/>, which discusses the use of Velocity in more detail.

- A better alternative is to use the `jQuery.Animate-Enhanced` plugin or the `animate` helper from `jQuery++`; both will convert animations to use CSS3 equivalents by default, where supported.

So, how do we handle animation requests on a responsive site when working with jQuery? There are several ways of doing this; let's explore this key question in more detail.

Handling animation requests on a responsive site

The best route to animate content within a responsive site when working with jQuery might actually seem a little perverse: don't use jQuery unless you absolutely have to! At this point, you may think I have completely lost the plot, but here are a few good reasons for this:

- jQuery is not optimized for animation; the line of demarcation between styles in the style sheet, HTML, and JavaScript will start to blur, which means that we lose control over how our content is styled.
- Animation doesn't work well on mobile devices when done with jQuery; to improve performance, additional CSS styling has to be used.
- We lose control over which rules are applied to specific elements due to CSS specificity – keeping styles within the CSS style sheet means that we can retain control.
- jQuery animations are resource-hungry by default. On a simple site, this will have a minimal impact, but on larger sites, the impact will be significantly higher.

- A bonus of using a pure CSS approach is that it allows you to make use of CSS preprocessors, such as **Syntactically Awesome Stylesheets (SASS)** or Less, to handle media queries. This form of shorthand CSS allows you to be more efficient at writing styles while still maintaining the final desired output.

With this in mind, let's take a look at a few pointers that we can use to handle animation requests on responsive sites:

- Think mobile first. If you're using CSS, then work on the basis of the smallest screen you want to accommodate first and then add additional media queries to handle changes to the layout when viewed on increasingly larger devices. Consider the use of a CSS Media Queries boilerplate, such as the one created by developer Paul Lund at <http://www.paulund.co.uk/boilerplate-css-media-queries>; we can then insert animation rules within each of the appropriate breakpoints.
- Avoid the use of the `.css` statement in your jQuery code, but use the `.addClass()` or `.removeClass()` methods instead – this allows you to maintain a separation of concerns, with a clear demarcation between the content and presentational layers. A good example of how this can be used (for those who are not sure) is given at the Animation Cheat Sheet site by Justin Aguilar, at <http://www.justinaguilar.com/animations/>. This produces a variety of different animations, all of which can be added using `.addClass()`.
- Work on the basis of using prefix-free versions of attributes in your code and then use an autoprefixer to add any vendor prefixes automatically, as needed. This becomes a cinch when using something to the likes of Grunt and a plugin, such as `grunt-autoprefixer`.
- Consider making use of the `jQuery.Animate-Enhanced` plugin (available at <https://github.com/benbarnett/jQuery-Animate-Enhanced>) where possible. Although it is a few years old, it still works with the current versions of jQuery; it extends `$.animate()` to detect transitions and replaces them with CSS equivalents.



Another plugin that is worth taking a look at is `Animsition`, available at <http://git.blivesta.com/animstion>.

- The trick here is to not rely on its use as a permanent part of the site but as a tool for replacing the existing jQuery animations with CSS equivalent styles. The more you can shift to using CSS, the less impact there will be on your pages, as demands for server resources will be reduced.

- Keep a close eye on <http://www.caniuse.com>. Although browser support for CSS3 transformations and transitions is very good, there are still a couple of instances where WebKit prefixes have to be used, namely for Safari and iOS Safari (mobile).
- Make use of `requestAnimationFrame` (and `clearAnimationFrame`) where possible within your animations. This will help conserve resources when animations are not visible. This will require the use of jQuery, but as we should aim to keep this for the most complex animations, the impact of using the library will be reduced.
- Take a look at sites such as <http://cssanimate.com/> – these allow you to generate complex keyframe-based animations that can be dropped into your existing code. If you have concerns that the existing content can't be animated, then it is possible that this site will help remove some of your doubt.
- Ask yourself this question: "If my animation is really complex, is it going to be effective?" Animations can be visually stunning if done well, but this does not mean that they need to be complex. Often, simple and well thought out animations work better than their complex, resource-hungry equivalents.

The important point to consider here is that using jQuery to animate content should not be completely discounted; with browser support for CSS animations continually evolving, it makes a strong case for using the latter as the basis for most animations.

The jQuery Team is conscious that jQuery was never designed for the efficient animating of content. There are ongoing discussions, at the time of writing this book, around the introduction of a version of Velocity.js; in principle, this will likely improve the effectiveness of using jQuery to animate content, but this is some way off becoming reality!

In the meantime, we should give careful consideration to the balance of jQuery versus CSS animation that is used, and aim to remove jQuery animations being used if CSS animations can be used in their place.



To help prove a point, Chris Coyier produced a CodePen example of how a reasonably simple site can be made responsive and can contain CSS-based animations, which you can view at <http://codepen.io/chriscoyier/pen/tynas>.

Okay, let's move on. We'll stay with the theme of animating, but this time we'll look at how we can achieve this on mobile devices. There are some considerations we need to be aware of; let's take a look at these in more detail.

Animating content for mobile devices

So far, we've considered the use of jQuery to animate content on responsive sites, but what about the mobile platform? There has been a significant increase in the use of non-desktop devices (such as laptops and smartphones) to view content. This brings some additional considerations that we need to make in order to make the most of performance on mobile devices.

Animating on a mobile platform is less about writing code but more about deciding which technologies to use; in most cases, simply writing jQuery code will work, but it won't be as effective as it should be.

The secret behind getting the best experience is in the use of the smartphone's **GPU** or **Graphics Processing Unit**; to do this, we can offload standard jQuery animations (which are slower) by enabling 3D rendering.



Although this browser should work on all desktop and mobile devices, you will get best results in a WebKit-based browser, such as Google Chrome.



Let's explore this in more detail with a simple example, which has 3D rendering enabled:

1. For this demo, we need three files. Go ahead and extract `mobileanimate.html`, `mobileanimate.css`, and `jquery.min.js` from the code download and save them in the relevant folders in your project area.
2. In a new file, add the following code. It handles the animation of our drop-down box. We'll go through it in detail, beginning with the assignment of a number of variables needed for our code:

```
var thisBody = document.body || document.documentElement,
    thisStyle = thisBody.style,
    transitionEndEvent = 'webkitTransitionEnd',
    transitionend',
    cssTransitionsSupported = thisStyle.transition !==
    undefined,
    has3D = ('WebKitCSSMatrix' in window && 'm11' in new
    WebKitCSSMatrix());
```

3. Next up comes the initial check that adds the `accordion_css3_support` class to the `ul` object, if the browser supports CSS3 transforms:

```
// Switch to CSS3 Transform 3D if supported & accordion
element exist
if(cssTransitionsSupported && has3D ) {
    if($('.children').length > 0) {
        $('.children').addClass("accordion_css3_support");
    }
}
```

4. The magic happens in this event handler. If CSS3 transitions are not supported, then the drop-down will use the `slideToggle` method to open or close; otherwise, it will use a CSS3 transform instead:

```
$('.parent a').on('touchstart click', function(e) {
    e.preventDefault();
    // If transitions or 3D transforms are not supported
    if(!cssTransitionsSupported || !has3D ) {
        $(this).siblings('.children').slideToggle(500);
    }
    else {
        $(this).siblings('.children').toggleClass("animated");
    }
});
```

5. Save the file as `mobileanimate.js`. If all went well, you will see a styled drop-down box ready to be opened, as shown here:

Demo: Replacing jQuery Animation With CSS Hardware Acceleration

[Rate the App ↗](#)

Try clicking on the drop-down arrow. At face value, it will appear that our drop-down is no different from any other; it expands and contracts in the same way as any other drop-down box. In reality, our code uses two important tricks to help manage animations; let's take a moment to go through the significance of both when working with jQuery.

Improving the appearance of animations

If we take a closer look at the code, there are two points of interest for us; the first is in the jQuery code:

```
if(!cssTransitionsSupported || !has3D ) {
    $(this).siblings('.children').slideToggle(500);
}
else {
    $(this).siblings('.children').toggleClass("animated");
}
```

The second is shown in two places in the CSS style sheet:

```
.accordion_css3_support { display: block; max-height: 0;
    overflow: hidden; transform: translate3d(0,0,0);
    transition: all 0.5s linear; -webkit-backface-visibility:
    hidden; -webkit-perspective: 1000; }
.children.animated { max-height: 1000px; transform:
    translate3d(0,0,0); }
```

"Why are these important?" I hear you ask. The answer is easy. In most cases, we will probably use the `slideToggle()` event handler. There's nothing wrong in this, except that the animation is not hardware-accelerated (and will also need you to convert it to CSS) and hence isn't going to make the best use of the platform's capabilities. In addition, it blurs the line between code and styles; it makes it harder to debug styles if we have them both in the code and in a style sheet.

A better alternative is to work out whether the browser supports CSS3 transforms (or similar) and to apply a new class that we can style within the style sheet. If the browser doesn't support transforms, then we simply fall back to using the `slideToggle()` method in jQuery instead. The benefit of the former is that CSS styles will reduce the resources required to run the animation and help conserve resources.



If jQuery must still be used, then it is worth testing the value set for `jQuery.fx.interval` – try somewhere around 12 fps to see whether this helps improve performance; more details are available in the main documentation at <http://api.jquery.com/jquery.fx.interval/>.

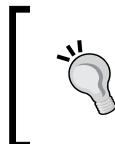
The second point of interest is perhaps a little less obvious; if we apply the transform `translate3d(0, 0, 0)` to any CSS rule that contains animations, then this is enough to enable 3D rendering and allows the browser to give a smooth experience, by offloading animations to the GPU. In some browsers (such as Google Chrome), we might get instances of flickering; we may need to add the following line of code to remove the unwanted flicker:

```
-webkit-backface-visibility: hidden; -webkit-perspective: 1000;
```

It is also possible that `translate3d(x, y, z)` doesn't enable hardware acceleration for some platforms (such as iOS 6); we can use `-webkit-transform: translate(0)` instead.

Ultimately, while there may be instances where we need (or prefer) to use jQuery to animate content, thought should be given as to whether it is really the right tool and whether CSS animations can be used in its place.

A good example of this is shown at JSFiddle (<http://jsfiddle.net/ezanker/Ry6rb/1/>), which uses the Animate.css library from Dan Eden to handle the animations, leaving jQuery as a dependency for jQuery Mobile that is used in the demo. Granted, the version of jQuery is a little old, but the principle is still very sound!



The Treehouse team posted a good blog entry that explores the science of how animations and transitions affect performance, which is worth a read; you can find it at <http://blog.teamtreehouse.com/create-smoother-animations-transitions-browser>.

Let's change focus and move on. Hands up if you've visited a site with a parallax scrolling effect? Parallax...scrolling...Not sure what it's all about? No problem, over the next few pages, we're going to take a look at what has become one of the hottest techniques in web design but can equally backfire if it is not properly implemented in our projects.

Implementing responsive parallax scrolling

What is parallax scrolling all about? Put simply, it involves moving the background at a slower rate than the foreground to create a 3D effect while you scroll down the page.

Originally created by Ian Coyle for Nike back in 2011, parallax scrolling is a popular technique to use. It can provide a subtle element of depth but can be equally overwhelming if you don't use it properly!

To get a flavor of what is possible, take a look at the article on the Creative Bloq website, at <http://www.creativebloq.com/web-design/parallax-scrolling-1131762>.

There are dozens of parallax scrolling plugins available, such as the parallax.js plugin from PixelCog (at <http://pixelcog.github.io/parallax.js/>) or Stellar.js by Mark Dalgleish, available at <http://markdalgleish.com/projects/stellar.js/>. Arguably, the most well-known plugin is Skrollr, which can be downloaded from <https://github.com/Prinzhorn/skrollr>—this will form the basis of our next demo.

Building a parallax scrolling page

If you spend any time to research on the Internet, you will no doubt come across lots of tutorials that cover adding a parallax scrolling effect to a site. Over the next few pages, we'll use a tutorial by the Australian frontend developer, Petr Tichy, as a basis for our next exercise. After all, there is no sense in trying to reinvent the wheel, right?



The original tutorial can be viewed at <https://ihatetomatoes.net/how-to-create-a-parallax-scrolling-website/>.

Our next demo will use the well-known Skrollr library (available at <https://github.com/Prinzhorn/skrollr>) to construct a simple page that scrolls through five images, but we'll also use a number of effects to control how the images scroll down the page:



Now that we've seen what our demo will produce, let's get stuck in by performing the following steps:

1. We'll begin by extracting the `parallax` folder from a copy of the code download that accompanies this book; save the entire folder to your project area.
2. We need a couple of additional plugins for our demo to work, so go ahead and download the following:
 - **ImagesLoaded**: <https://raw.githubusercontent.com/desandro/imagesloaded/master/imagesloaded.pkgd.js>; save the file as `imagesloaded.js`
 - **Skrollr**: <https://raw.githubusercontent.com/Prinzhorn/skrollr/master/src/skrollr.js>
 - **ViewPortSize**: <https://github.com/tysontanich/viewportSize>

Save all of these plugins in the `js` subfolder within the `parallax` folder.

3. In a new file, add the following code; this handles the initialization of the Skrollr plugin. Let's go through it in detail, beginning with a `ready` DOM statement that sets up a number of variables and then preloads the images using the ImagesLoaded plugin before resizing them and fading in each section:

```
$(document).ready(function($) {  
    // Setup variables  
    $window = $(window);  
    $slide = $('.homeSlide');  
    $slideTall = $('.homeSlideTall');  
    $slideTall2 = $('.homeSlideTall2');  
    $body = $('body');  
  
    //FadeIn all sections  
    $body.imagesLoaded( function() {  
        setTimeout(function() {  
            // Resize sections  
            adjustWindow();  
  
            // Fade in sections  
            $body.removeClass('loading').addClass('loaded');  
        }, 800);  
    });
```

4. Immediately below the DOM function and before the closing brackets, add the following code. This handles the resizing of each of the slides to the appropriate window height or to a minimum height of 550px, whichever is greater:

```
function adjustWindow(){  
    var s = skrollr.init(); // Init Skrollr  
    winH = $window.height(); // Get window size  
  
    // Keep minimum height 550  
    if(winH <= 550) { winH = 550; }  
  
    // Resize our slides  
    $slide.height(winH);  
    $slideTall.height(winH*2);  
    $slideTall2.height(winH*3);  
  
    // Refresh Skrollr after resizing our sections  
    s.refresh($('.homeSlide'));  
}
```

5. If all is well, when you preview the results, images will cross from one to another when we scroll up or down, as shown in this screenshot:



Parallax scrolling as a technique can produce some really stunning effects when used well. For some great examples, take a look at Costa Coffee's site, at <http://www.costa.co.uk>, or Sony's Be Moved site, at <http://www.sony.com/be-moved/>. It's hard to believe that such original designs are based on parallax scrolling!



Take a look at one of Petr's tutorials on how to make parallax scrolling responsive, at <https://ihatetomatoes.net/make-parallax-website-responsive/>.



Considering the implications of parallax scrolling

Although it may be hard to believe that such beautiful sites can be created using parallax scrolling, this must be tempered with a warning: this technique does not come without its issues. Granted, most (if not all) can be overcome with some care and attention; nevertheless, these issues can trip up any designer if care is not taken over the design and implementation. Let's explore some of these issues in more detail:

- The biggest killer is that parallax scrolling is not SEO-friendly by default. There are techniques available to get around this, such as jQuery or multiple pages, but they will impact analytics or server resources. The digital marketing strategist Carla Dawson has written an excellent article that discusses the merits of these workarounds, which is available at <http://moz.com/blog/parallax-scrolling-websites-and-seo-a-collection-of-solutions-and-examples>—it is worth a read!
- Parallax scrolling will (naturally) require visitors to scroll; the key here is to ensure that we are not creating single pages that scroll for too long. This might have an impact on performance for mobile users and put visitors off.
- The use of jQuery to create effects based on this technique can itself be a drawback; jQuery will have an impact on page loading times, as the position of each element on the page has to be calculated. We can mitigate against this to a degree, by customizing our copy of jQuery using the techniques we covered back in *Chapter 1, Installing jQuery*, but there will always be an element of reduced performance when using the library.
- Parallax scrolling can reveal a number of usability issues. The layout can appear haphazard to end users, if the balance of visual appeal against content and ease of access is not even. Parallax scrolling will be suitable instances where you might expect visitors to browse your site once, or for a company to show case what they can do—it can be harmful for those situations where you are pitching for a product or business.
- In a number of cases, you will find that parallax scrolling doesn't work on mobile devices; this is largely due to how animations are executed at the end, which breaks parallax scrolling. Attempts have been made to work around this, with varying levels of success. The following are a couple of examples of successful attempts:
 - Using the Stellar.js jQuery parallax plugin, which is available at <http://markdalgleish.com/projects/stellar.js/>; this in tandem with the Scrollability plugin, from <http://joehewitt.github.com/scrollability/>, can be used to produce a touch-friendly parallax scrolling effect. The plugin works both in desktop and mobile browsers, so consideration should be given to checking for touch support and switching methods, as appropriate. The plugin author Mark Dalgleish explains how to achieve this using iScroll.js at <http://markdalgleish.com/presentations/embracingtouch/>.
 - A pure CSS version by Keith Clark is available at <http://codepen.io/keithclark/pen/JycFw>—he explains the principles used in detail on his site, at <http://keithclark.co.uk/articles/pure-css-parallax-websites/>.

The key message for parallax scrolling is to not rush in; it's true that there are some sites that have managed to create some stunning examples of parallax scrolling, but a lot of thought and planning will have gone into building the example so that it is performant, caters to SEOs, and still presents a usable experience to the visitor.

Summary

Animating content within projects can be very satisfying if done well; this relies on us not just using the right code but also deciding whether jQuery is the right tool or whether CSS animations will be preferable for our needs. We've covered a lot over the last few pages, so let's take a moment to recap what we learned.

We kicked off with a discussion on the merits of using jQuery or CSS and when it might be preferable to use one instead of the other; we saw some of the benefits of using CSS and that circumstances may dictate the use of jQuery.

We then moved on to take a look at the classic issue that besets jQuery developers at some point in their lives, namely controlling the animation queue; we saw how to implement a quick and dirty fix, with subsequent improvements to reduce or eliminate the issue.

Next up came a discussion on the use of easing functions; we saw how easy it is to not just rely on tried and tested sources such as the jQuery UI but also to develop simple actions that extend core jQuery. We took a look at building our own custom-easing functions followed by converting those that we might see in CSS to jQuery equivalents.

We then rounded out the chapter with a look at some animation examples in the form of animating buttons, implementing an overlay effect with a twist and animating content on a responsive site.

In the next chapter, we're going to take a look at advanced event handling. In most cases, people use `.on()` or `.off()`, but as we'll see, this only scratches the surface of what is possible with jQuery.

7

Advanced Event Handling

How many times do you go to a website to perform an action? It might be online banking, or perhaps purchasing something from Amazon; in both cases, the sites will detect the actions taking place, and respond accordingly.

Part of working with jQuery is knowing how and when to respond to different types of events. In most cases, people are likely to use the `.on()` or `.off()` event handlers to handle them. While this works perfectly well, it just scratches the surface of what can be done with event handling. In this chapter, we will take a look at some of the tips and tricks we can use to expand our skills when it comes to event handling. We will cover the following topics:

- Delegating events
- Using the `$.proxy` function
- Creating and decoupling custom event types
- Namespacing events

Intrigued? Let's get on with it then!

Introducing event handling

A question – how often do you go online to perform a task? I'll bet it's a fair few times a week; it could be anything from online banking, to hitting Amazon to get that latest DVD (DVDs – who downloads them, I wonder?)

That aside, we can't escape having to click on a link or a button to advance through a process. In most cases, the code behind the event is likely to be the ubiquitous click handler, or it could even be `.change()` or `.hover()`. All are shorthand forms of the `.on()` (or even `.off()`) event handlers, and are of course functionality equivalent to something like the following:

```
$('a').on('click', function() {
    $(this).css('background-color', '#f00');
});
```

This will turn the selected element to a nice shade of red. However, there is more to event handling than simply defining an action on a known element. Over the next few pages, we're going (to quote a nautical term) to push the boat out, and take a look at a few tips and tricks that we can use, to help develop our skills further. We'll begin with a look at event delegation.

Delegating events

Someone once said that the art of being a good manager is to know when to delegate. I hope that this wasn't an excuse for them to offload a horrible job to a subordinate, although the cynical might say otherwise!

Leaving aside the risk, delegation follows the same principles in jQuery. If we need to create an application which requires binding some form of event handler to lots of elements of the same type, then we might consider writing event handlers to cover each element.

It'll work to an extent, but is very wasteful of resources. If the list is large, then events will be bound to all of the elements within, which uses more memory than is needed. We can get around this by using **event delegation**, where we can shift to binding one event handler to a single ancestor element that serves multiple descendants, or enable event handling for newly created elements.

There are a few tricks we can use to help us with better managing of events using delegation. Before we take a look at them, let's quickly recap the basics of how event delegation works.

Revisiting the basics of event delegation

A question – how often have you used `.on()`, or even `.off()` when coding event handlers in jQuery? I'll bet the answer is probably countless times. If you've not already used event delegation before, then you're already halfway to using it without realizing it!

Event delegation relies on the use of **event propagation**, or event bubbling as it is sometimes known. It is the key to understanding how delegation works. Let's work through a quick example.

Imagine we're using the following HTML code as the basis for a list:

```
<div id="container">
  <ul id="list">
    <li><a href="http://domain1.com">Item #1</a></li>
    <li><a href="/local/path/1">Item #2</a></li>
    <li><a href="/local/path/2">Item #3</a></li>
    <li><a href="http://domain4.com">Item #4</a></li>
  </ul>
</div>
```

Nothing outrageous here – it's a simple example. Any time one of our anchor tags is clicked, a click event is fired for that anchor. The event is dispatched in one of the three phases: **capturing**, **target**, and **bubbling**.

It will be captured at the document root, work its way down until it hits its target (The `li` tag), before bubbling back up to the document root, as shown next:

- document root
- `<html>`
- `<body>`
- `<div #container>`
- `<ul #list>`
- ``
- `<a>`

Yikes! This means that each time we're clicking on a link, we're effectively clicking on the whole document! Not great! It's expensive on resources, and even if we were to add additional list items using code such as this:

```
$("#list").append("<li><a href='http://newdomain.com'>Item #5</a></li>");
```

We would find that the aforementioned click handler wouldn't work with these items.



The bubbling example used here is somewhat simplified, and doesn't show all the various phases. For a useful discussion, head over to the comments posted on Stack Overflow at <http://stackoverflow.com/questions/4616694/what-is-event-bubbling-and-capturing>.

Reworking our code

Instead of adding a directly bound handler, we can take advantage of event propagation, and rework our handler to listen for **descendant** anchors, instead of binding to existing anchor tags only. This can be seen in the following code:

```
$("#list").on("click", "a", function(event) {  
    event.preventDefault();  
    console.log($(this).text());  
});
```

The only difference in the code is that we've moved the `a` selector to the second parameter position of the `.on()` method. This creates a single event handler against `#list`, with the event bubbling up one level from `a` to `#list`. Event delegation removes the need to create multiple event handlers, which is wasteful - the code will work equally well with both existing anchor tags within `#list`, and with any that are added in the future.



If you would like to learn more about event delegation, then it is worth viewing the jQuery API documentation, which is at <http://learn.jquery.com/events/event-delegation/>. The jQuery documentation also has a useful section on using `.on()` within delegated events at <http://api.jquery.com/on/>.

Supporting older browsers

A small point – if you need to rework older code, then you may see `.bind()`, `.live()`, or `.delegate()` as event handlers. All were used to delegate events prior to jQuery 1.7, but should now be replaced with `.on()`. In fact, the first, `.bind` is a one line function that calls to `.on` (and its partner, `.off()`):

```
7491     bind: function( types, data, fn ) {  
7492         return this.on( types, null, data, fn );  
7493     },  
7494     unbind: function( types, fn ) {  
7495         return this.off( types, null, fn );  
7496     },
```

The same applies for `.delegate()` and its partner event handler, `.undelegate()`:

```

7498     delegate: function( selector, types, data, fn ) {
7499       return this.on( types, selector, data, fn );
7500     },
7501     undelegate: function( selector, types, fn ) {
7502       // ( namespace ) or ( selector, types [, fn] )
7503       return arguments.length === 1 ? this.off( selector, "*" ) :
7504         this.off( types, selector || "**", fn );
7505     }

```

It should be noted that `.on()` mimics the behaviors found when using `.bind()` or `.delegate()`. The former is very resource hungry as it attaches to every single element it can match; the latter still has to work out which event handler to invoke. However, the scope of this should be smaller in comparison to using the `.bind()` method.

Now that we've delved into the inner workings of `.on()`, let's put it into action, and create a simple demo to remind ourselves of how delegation works within jQuery.

Exploring a simple demonstration

It's time for a little action, so let's start with a quick reminder of how event delegation works, when using jQuery:

1. Let's start by extracting the files we need from the code download that accompanies this book. For this demo, we need the `simpledelegation.html`, `simpledelegation.css`, and `jquery-ui.min.css` files.
2. Save the CSS files within the `css` subfolder of our project area. The HTML markup needs to be stored in the root area of the project folder.
3. In a new file, add the following code—save the file as `simpledelegation.js`, and store it in the `js` subfolder of our project area:

```

$(document).ready(function(event) {
  var removeParent = function(event) {
    $('#list').parent('li').remove();
  }

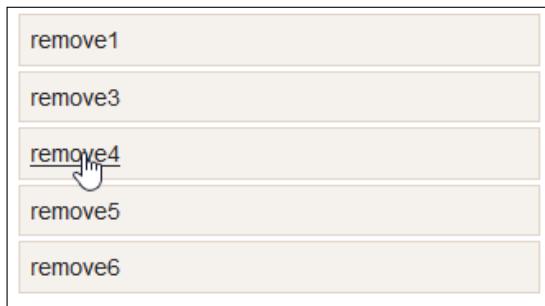
  var removelistItem = function(event) {
    $(event.target).parent().remove();
  }

  $('li.ui-widget-content').children().on("click",
    removeParent);

  $('ul').on("click", "li", removelistItem);
});

```

4. If all is well, we should see the following list of items, when previewing the results in a browser:



5. Try clicking on a number of the links – if you click on any of the remove links, then the list item will be removed; clicking on one of the list items will remove all of the items from the list.

The key to this demo is the following one line:

```
$('ul').on("click", "li", removelistItem);
```

Although we have multiple items within the list, we've created one single delegated event handler. It bubbles up to the parent of the `` item we clicked, then removes it. In this instance, we've separated out the function that is called when the event is triggered; this could easily have been incorporated into the handler.

Now that we've revisited the basics of event delegation, let's take a look at some of the reasons why event delegation can lead to increased performance, when working with a lot of similar elements.

Exploring the implications of using event delegation

The key benefit of implementing delegated events in place of direct equivalents, is reducing memory usage and avoiding memory leaks if multiple event handlers are present in our code. Normally, we would need to implement an event handler for each instance where we need something to happen.



The real impact of using event delegation is around the savings in memory usage, gained from where event handler definitions are stored within the internal data structure.



Instead, reducing the number of event handlers means that we can reduce memory leaks and improve performance (by reducing the amount of code that has to be parsed). As long as we are careful about where we bind the event handler, there is a potential to dramatically reduce the impact on the DOM and the resulting memory usage, particularly in larger applications. The bonus is that if event delegation has been implemented, it will apply equally to existing elements that have been defined, as well as those that have yet to be created. Directly applied event handlers will not work; they can only be applied to elements that already exist prior to the event handler being called in the code.

The ability to handle events that exist, and those that have yet to happen, sounds like a good thing. After all, why repeat ourselves, if one event handler can handle multiple events, right? Absolutely – as long as we manage it carefully! If we trigger an event on a specific element, such as an anchor tag for example, then this will be allowed to handle the event first. The event will bubble up until it reaches document level, or a lower event handler decides to stop event propagation. This last part is key – without control, we could end up with unexpected results, where event handlers have responded, or not fired, contrary to expectations.



To see a detailed explanation of what can happen, take a look at <http://css-tricks.com/capturing-all-events/>. It contains links to examples on CodePen that illustrate this issue very well.

To help reduce the impact of event bubbling causing event handlers to fire out of turn, we use methods such as `event.stopPropagation()`. This is not the only trick we can use, so let's take a moment to explore some of the options available when using event delegation.

Controlling delegation

Taking advantage of event bubbling increases the scope for reducing the number of event handlers we need to implement within our code; the downside is the instances of unexpected behavior, where event handlers may not be triggered at the desired points.

To control which elements might trigger a delegated event handler, we can use one of the following two tricks: `event.stopPropagation()`, or trapping the event target and determining if it matches a given set of conditions (such as a specific class or `data-name`).

Let's take a look at this second option first – an example block of code might look like the following:

```
$( "ul.my-list" ).click(function(event) {  
    if ( $( event.target ).hasClass("my-item") ) {  
        handleListItemAction(event.target);  
    }  
    else if ( $( event.target ).hasClass("my-button") ) {  
        handleButtonClickedAction(evt.target);  
    }  
});
```

That's one clumsy way of doing things! Instead, we can simply instigate a check on the class name, using a variation of the delegated event handler, as shown next:

```
$( "ul.my-list" ).on("click", ".my-item", function(evt) {  
    //do stuff  
});
```

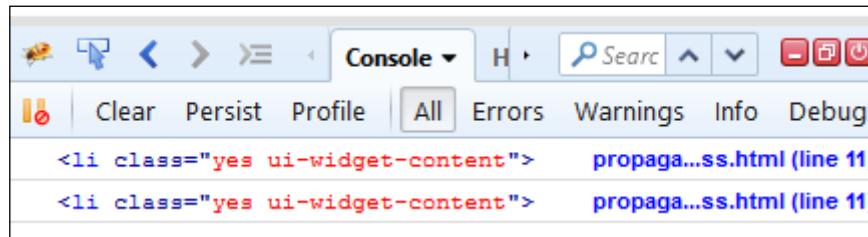
This is a really simple trick we can use – it's so simple, it probably doesn't count as a trick, as such! To see how easy it is to make the change, let's run through a quick demo now:

1. From the code download, we need to extract the `propagation-css.html` and `propagation.css` files. These contain some simple markup and styles for our basic list. Save the CSS file in the `css` subfolder of our project area, and the HTML markup at the root of the same area.
2. Next, we need to create the event handler that will fire when the conditions match. Go ahead and add the following to a new file, saving it as `propagation-css.js` in the `js` subfolder of our project area:

```
$(document).ready(function() {  
    $('#list').on('click', '.yes', function eventHandler(e) {  
        console.log(e.target);  
    });  
});
```

At this point, if we preview the results in a browser, we will have a simple list, where list items darken if we hover over a specific item. Nothing particularly special about this – it's just borrowing some styling from jQuery UI.

However, if we fire up a DOM inspector, such as Firebug, and then hover over each item, we can see console output is added each time we hover over an item with a class of `.yes`:



So, instead of providing a selector as we did back in *Exploring a simple demonstration*, we simply used a class name; the event handler function will only fire if it matches the specified class name.

We can even apply a `data-` tag as our check:



```
$(document).on('keypress', '[data-validation="email"]',
  function(e) {
    console.log('Keypress detected inside the element');
})
```

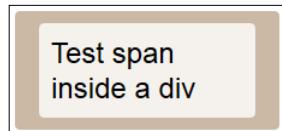
Using the `stopPropagation()` method as an alternative

As an alternative, we can use an all-jQuery solution in the form of `stopPropagation()`. This prevents the event from bubbling up the DOM tree, and stops any parent handlers from being notified of the event.

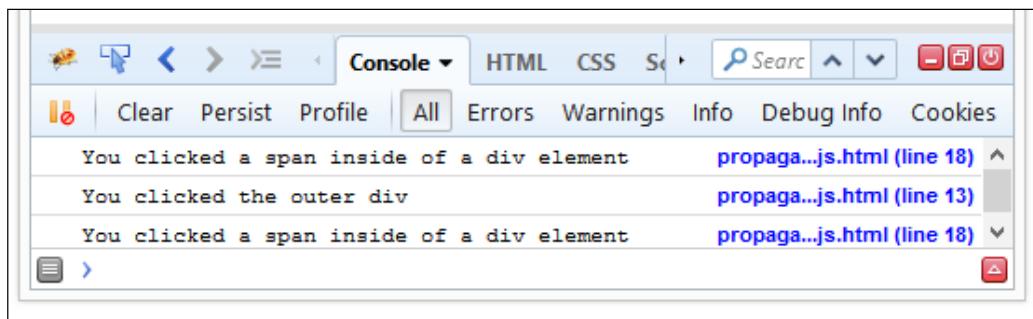
This one-liner is a breeze to implement, although the key to using it is ensuring we add it at the right point in our code. If you've not used it before, then it needs to go within the event handler, immediately after the last command in the handler (as highlighted in the following snippet):

```
document.ready(function ($) {
  $('div').on("click", function (event) {
    console.log('You clicked the outer div');
  });
  $('span').on("click", function (event) {
    console.log('You clicked a span inside of a div element');
    event.stopPropagation();
  });
})
```

As a quick check, try extracting the propagation-.js files from the code download that accompanies this book. Save them in the relevant folders within our project area. If we preview them in a browser, we'll see a simple **span** enclosed within a **div**. Refer to the following image:



The key to this demo lies within the DOM Inspector. Try clicking on the grey-brown outer ring, or the span within it, and we will see the results of what we've selected appear in the console log, as shown next:



If you comment out the `event.stopPropagation()` line within the code, the click event attached to `div` will also be invoked.



Event propagation should not be stopped unless necessary. There is a useful article at <https://css-tricks.com/dangers-stopping-event-propagation/> which discusses the problems you might encounter if propagation is stopped.

Okay, let's change focus and switch to another key concept within event handling. It's time to take a look at using the `$.proxy` function, and why this is needed, if event delegation doesn't propagate sufficiently for our needs.

Using the `$.proxy` function

Up until now, we've covered how making use of event bubbling can help us reduce the need for lots of event handlers; provided we manage the bubbling carefully, then delegation can prove a very useful tool in developing with jQuery.

The flipside of this is that in some instances we may need to give jQuery a helping hand; when it doesn't propagate sufficiently high enough up the chain! At first this may not make sense, so let me explain what I mean.

Let's, for argument sake, imagine we have an event handler that has been created as an object, and that we want to call it when clicking on a link:

```
var evntHandlers = {
    myName : 'Homer Simpson',
    clickHandler : function(){
        console.log('Hello, ' + this.myName);
    }
};

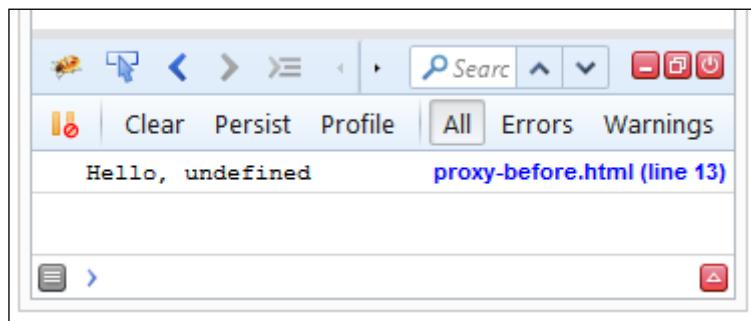
$("a").on('click',evntHandlers.clickHandler);
```

If we ran this in a browser, what would you expect to see in the console log area?



To find out, try extracting the `proxy-before.html` file from the code download that accompanies this book. Make sure you have a DOM inspector installed!

If you were expecting to see **Hello, Homer Simpson**, then I will have to disappoint you; the answer won't be what you expect, but instead will be **Hello, undefined**, as shown in the following image:



Okay, so what gives?

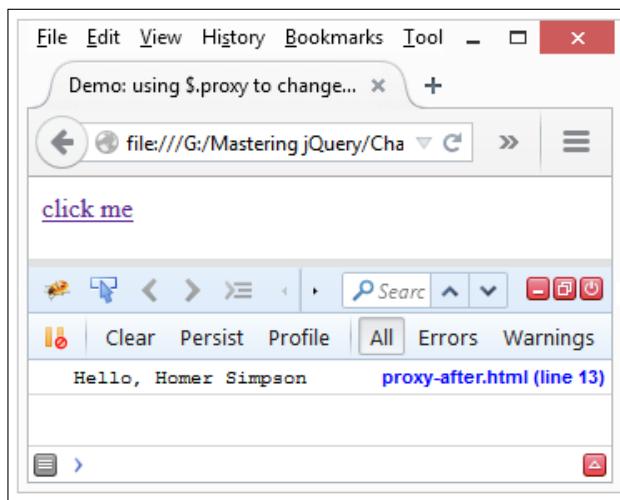
The reason for this is that the context being used is within the `clickHandler` event, and not the `evntHandler` object; we don't have a `myName` property within the `clickHandler` event.

Thankfully, there is a simple fix for this. We can use `$.proxy` to force a change of context, as shown next:

```
var evntHandlers = {
    myName : 'Homer Simpson',
    clickHandler : function(){
        console.log('Hello, ' + this.myName);
    }
};

$("a").on('click',$.proxy(evntHandlers.clickHandler,evntHandlers));
```

To see this in action, extract the `proxy-before.html` and `proxy-after.html` files from the code download that accompanies this book. If you run them in a browser, you will see the same results as shown in the following screenshot:



This is a simple change to make, but it opens up a wide variety of possibilities. It is a shorthand method of setting the context for a closure. We could of course use the plain JavaScript `.bind()` methods. Instead, using `$.proxy` ensures that the function passed in is actually a function, and that a unique ID is passed to that function. If we add namespaces to our events, we can be sure that we unbind the correct event. The `$.proxy` function is seen as a single function within jQuery, even if it is used to bind different events. Using a namespace rather than a specific proxied function will avoid unbinding the wrong handler in our code.



If you would like to learn more about using `$.proxy`, then it is worth reading the documentation on the main jQuery site, which is available at <http://api.jquery.com/jquery.proxy/>.

To give us a real flavor of what is possible, consider this for a moment: how many times have you ended up with functions nested three to four levels deep? Consider the following code:

```
MyClass = function () {
    this.setupEvents = function () {
        $('a').click(function (event) {
            console.log($(event.target));
        });
    }
}
```

Rather than working with the above mentioned code, we can refactor it to increase readability, by using `$.proxy`, as shown next:

```
MyClass = function () {
    this.setupEvents = function () {
        $('a').click( $.proxy(this, 'clickFunction'));
    }

    this.clickFunction = function (event) {
        console.log($(event.target));
    }
}
```

I think you will agree that this is much easier to read, right?

Okay – let's move on. I'm sure we are all familiar with creating event handlers in jQuery. However, chances are that you're working with standard event handlers. These will work perfectly well, but we're still limited in what we can do.

Well, let's change that. Using jQuery, we can create custom events that break the otherwise familiar mould of what we know is possible, and will allow us to create all kinds of event handlers. Let's take a look at how we can do this in action.

Creating and decoupling custom event types

If you've spent any time developing jQuery, then I am sure you are more than familiar with the standard event types that we can use, such as `.click()`, `.hover()`, or `.change()`.

These all serve a useful purpose, but all have one thing in common – we're a bit limited in what we can do with them! Our code will be dictated by the extent of what these handlers can do. What if we can break this limitation, and create *any* type of custom event handler?

Of course, we can always combine multiple events together, to be served by the same function:

```
$('input[type="text"]').on('focus blur', function() {
    console.log( 'The user focused or blurred the input' );
});
```

But this is still limited to those event handlers that are available out of the box. What we need is to break the mould and get creative in designing our own handlers.

No problem – we can use jQuery's special event functionality to build pretty much any type of event to suit our needs. This opens up a real world of possibilities, which might warrant a book in its own right. Over the next few pages, we'll cover a few of the concepts to help get you started on the right path to creating events.



For a more in-depth look at creating custom events, there is a useful article on the learn jQuery site, at <http://learn.jquery.com/events/introduction-to-custom-events/>.

The great thing about events is that they act just like their standard cousins, including bubbling up the DOM:

```
$( 'p' ).bind( 'turnGreen', function() {
    $(this).css('color', '#00ff00');
});

$( 'p:first' ).trigger( 'turnGreen' );
```

So, what goes into the makeup of a special event? Special events will often take the form of a plugin; the format may be similar, but we'll frequently see any one of the several `fixHooks`, which we use to control the behavior of event processing in jQuery.



jQuery special event hooks are a set of per-event-name functions and properties that allow code to control the behavior of event processing within jQuery.



Let's take a moment to have a look at the typical makeup of a special event plugin, before diving into an example of such a plugin.

Creating a custom event

The fixHooks interface provides a route to normalize or extend the event object that will override a native browser event. We might typically see a format such as the following used in our event plugin:

```
jQuery.event.special.myevent = {  
    noBubble: false,  
    bindType: "otherEventType",  
    delegateType: "otherEventType",  
    handle: function ($event, data {  
        // code  
    },  
    setup: function( data, namespaces, eventHandle ) {  
        // code  
    },  
    teardown: function( namespaces ) {  
        // code  
    },  
    add: function( handleObj ) {  
        // code  
    },  
    remove: function( handleObj ) {  
        // code  
    },  
    _default: function( event ) {  
        // code  
    }  
};
```

It's worth noting that when creating special event types, there are two methods that we will use frequently - `.on()`, for binding events, and `.trigger()`, for manually firing a specific event when needed. In addition, a special event plugin will expose a number of key methods which are useful to learn. Let's explore these for a moment:

Name of method / property	Purpose
<code>noBubble: false</code>	Boolean set to <code>false</code> by default. Indicates whether bubbling should be applied to this event type if the <code>.trigger()</code> method is called.
<code>bindType</code>	When defined, these string properties specify that a special event should be handled like another event type until the event is delivered. Use the <code>bindType</code> for directly attached events, and <code>delegateType</code> for those that have been delegated. In both cases, these should be standard DOM types, such as <code>.click()</code> .
<code>handle: function(event: jQuery.Event, data: Object)</code>	Calls a handle hook when the event has occurred, and jQuery would normally call the user's event handler specified by <code>.on()</code> or another event binding method.
<code>setup: function(data: Object, namespaces, eventHandle: function)</code>	Called the first time an event of a particular type is attached to an element. This provides the hook an opportunity to do processing that will apply to all events of this type on this element.
<code>teardown: function()</code>	Called when the final event of a particular type is removed from an element.
<code>add: function(handleObj)</code> <code>remove: function(handleObj)</code>	Called when an event handler is added to an element through an API such as <code>.on()</code> , or removed when using <code>.off()</code> .
<code>_default: function(event: jQuery.Event, data: Object)</code>	Called when the <code>.trigger()</code> or <code>.triggerHandler()</code> methods are used to trigger an event for the special type from code, as opposed to events that originate from within the browser.

It's worth getting to know these methods well, particularly if you use jQuery Mobile in your development. Mobile has a dependency on special events, to produce events such as `tap`, `scrollstart`, `scrollstop`, `swipe`, or `orientationchange`.



For more details on each method, take a look at the Gist by Ben Alman, which is available at <https://gist.github.com/cowboy/4674426>.

Special events will require a deeper level of knowledge, if you are using them to override standard behavior of events such as click or mouseover. To understand more of the inner workings, it is worth reading the article on the jQuery Learning Site at <http://learn.jquery.com/events/event-extensions/>. Note though – it will get quite complex!

Now that we've seen some of the inner workings of a special event plugin, it's time to get stuck in and see something in action. For this, we're going to use the jQuery Multiclick plugin produced by James Greene, to show how easy it is to capture an action such as triple-clicking, and use it to perform an action.

Working with the Multiclick event plugin

Creating a custom event can be as simple or as complex as is needed. For this demo, we're going to use the jQuery Multiclick event plugin by James Greene. The plugin is available from <http://jamesmgreene.github.io/jquery.multiclick/>. We'll use it to post some messages on screen, with the message changing on every third click. Refer to the following image:



Let's take a look at what is involved:

1. Let's start by extracting the following files from the code download that accompanies this book. For this demo, we'll need the `jquery.multiclick.js`, `jquery.min.js`, `multiclick.css`, and `multiclick.html` files. Store each of the files in the relevant subfolder within our project area.

2. In a new file, add the following code, saving it as `multiclick.js`:

```
$(document).ready(function() {
    var addText = "Click!<br>";
    var addBoom = "Boom...!<br>";

    $("button").on("click", function($event) {
        $("p").append(addText);
    });

    $("button").on("multiclick", { clicks: 3 }, function($event)
    {
        $("p").append(addBoom);
    });
});
```

3. This is required to configure the multiclick plugin, and trigger the appropriate responses when the mouse has been clicked.
4. Try running the demo in a browser. If all is well, we should see something similar to the screenshot shown at the start of the exercise, once we've clicked on the **Click me!** button a few times.

Although it probably has to be said that this isn't entirely representative of a real-world example, the techniques involved are nonetheless the same. The plugin is bound to the standard click handler, and will fire if the number of clicks reached is a multiple of the value stated in the configuration options for the plugin.

Namespacing events

So far, we've seen how we can delegate events and create handlers that can take custom triggers. These methods are perfect if we have a single click event handler, but what happens if we need to have multiple click handlers, for example?

Well, fortunately there's a simple solution: add a namespace to the event! Rather than talk about how it works, let's take a quick look at the following example:

```
$("#element")
.on("click", doSomething)
.on("click", doSomethingElse);
```

This code is perfectly acceptable – nothing wrong with this at all. Sure, it might not be quite as readable as some might like, but we're not worried about that – at least not for now!

The critical point here is if we were to call:

```
$("#element").off("click");
```

Then we would lose not only the first click handler, but the second one as well. This is not ideal. We can fix this by adding a namespace or identifier to the command, as shown next:

```
$("#element")
  .on("click.firsthandler", doSomething)
  .on("click.secondhandler", doSomethingElse);
```

If we run the same `.off` command now, then clearly neither event handler will be removed. But – suppose we make the following change:

```
$("#element").off("click.firsthandler");
```

Now we can safely remove the first event handler, without removing the second.

 If we had written `$("#element").off(".firsthandler")` instead, then it would have removed all event handlers that had this namespace assigned to them. This can be very useful when developing plugins.

The best way to understand how this works, is to see it in action. Let's take a look at the following simple example now:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>Demo: Adding namespaces to jQuery events</title>
  <script src="js/jquery.min.js"></script>
</head>
<body>
  <button>display event.namespace</button>
  <p></p>
  <script>
    $("p").on("test.something", function (event) {
      $("p").append("The event namespace used was: <b>" +
        event.namespace + "</b>") ;
    });
  </script>
</body>
</html>
```

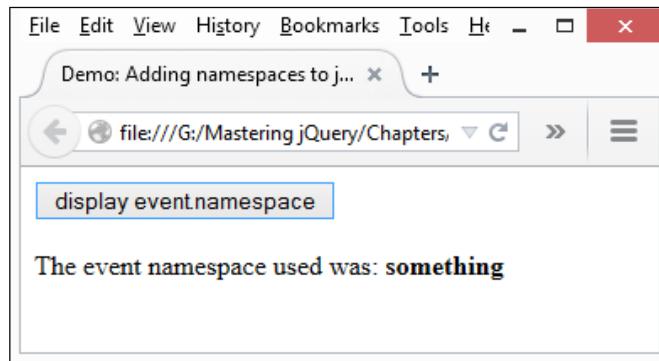
```
$("button").click(function(event) {
    $("p").trigger("test.something");
});
</script>
</body>
</html>
```



The code for this demo is available in the code download that accompanies this book, as the `namespacing.html` file. You will need to extract it and a copy of jQuery in order to run the demo.



Here, we've assigned two resize functions. We then remove the second using the namespace, which will leave the first completely untouched, as displayed in the following image:



If we use a DOM Inspector to inspect the code, we can clearly see the namespace being assigned; to do so, set a breakpoint on line 12, then expand the list on the right, as shown in the next image:

isTrigger	3
namespace	"something"
namespace_re	RegExp /(^ \.)something(\. \$)/
result	undefined

At first, this may seem like a really simple change, but I am a great believer in the phrase KISS - you get the idea!



There is no limit to the depth or number of namespaces used; for example, `resize.layout.headerFooterContent`. Namespaces can equally be used with standard event or custom event handlers.

Adding a namespace identifier is a really quick and easy fix that we can apply to any event handler. It gives us perfect control over any event handler, particularly when assigning functions to multiple instances of the same event type within our code.



If you are frequently creating complex event handlers, then it may be worth to take a look at the Eventralize library by Mark Dalgleish, which is available from <http://markdalgleish.com/projects/eventralize/>. Note, though, it hasn't had any updates for 2-3 years, but may be worth testing it to see if it helps consolidate and simplify your events.

Summary

Event handling is key critical to the success of any website or online application. If we get it right, it can make for an engaging user experience; getting it wrong can lead to some unexpected results! Over the last few pages, we've looked at few concepts to help develop our event handling skills; let's take a moment to review what we've learnt.

We kicked off with a quick introduction into event handling, before moving swiftly onto exploring event delegation as one tool where we can benefit from its use in our code. We first looked at the basics of event delegation, before examining the implications of using it, and learning how we can control it within our code.

Next up came a look at `$.proxy`, where we saw how jQuery sometimes needs a helping hand to ensure that an event is fired within the right context if our code means it doesn't propagate sufficiently high enough up the chain.

We then turned our attention to a brief look at creating custom event types and handlers, before exploring how such event handlers are constructed. We then used the jQuery Multiclick plugin as an example of how we can create these custom event handlers, before rounding up the chapter with a look at using namespacing to ensure that we can bind or unbind the right event handler in our code.

In the next chapter, we'll be looking at some of the visual ways we can enhance our sites – we'll see how applying effects, and managing the resultant effects queue can help either make or break the success of our sites.

8

Using jQuery Effects

Adding event handlers to any website is a necessary must; after all, we need some way to respond to legitimate events in our code.

The flip side of this is adding effects – done well, they can be hugely rewarding, although some of the novelty can wear off, particularly if you've used all of the core effects to death! Revitalize your sites with new, custom effects – we'll see how to do this in this chapter, as well as managing the resulting queues. Over the next few pages, we'll cover the following topics:

- Revisiting basic effects
- Adding callbacks
- Constructing custom effects
- Creating and managing the effect queue

Intrigued? Let's make a start...

Revisiting effects

A question – how many times have you visited a site to see content smoothly slide up, or gradually fade to nothing?

I'm sure that you will of course recognize these as effects provided in code; these can be anything from a simple slide up, to content appearing to fade from one image or element to another.

Creating effects is a key consideration of any website. We've already touched on some methods earlier in the book in *Chapter 6, Animating with jQuery*. I'm sure we're all familiar with the basic code for fading or toggling elements. No doubt you will have used codes such as `$("blockquote").fadeToggle(400);` or `$("div.hidden").show(1250);` countless times when developing websites.

Looks familiar? Over the next few pages, we'll touch on some additional tricks we can use to help push out the boat when it comes to adding effects, as well as considering some of the implications of using jQuery to provide these effects. Before we do so, there is an important consideration we need to clear up, which is to explore the key differences between simple animation and adding effects to elements.

Exploring the differences between animation and effects

Some of you may think that we've covered the provision of effects when we touched on animating back in *Chapter 6, Animating with jQuery*. It is true that there is some cross-over; a quick look at the API list for jQuery Effects will show `.animate()` as a valid effects method.

However, there is an important distinction – the content we've already covered is about *moving* elements; providing effects will focus on controlling the *visibility* of content. The great thing though, is that we can link the two together. `.animate()` can be used to implement both movement and effects within code.

Now that little distinction has been cleared up, let's get into some action. We'll start with a look at adding custom easing functions to our effects.

Creating custom effects

If you've spent any time applying effects to animated elements, then you will very likely have used `.animate()`, or one of the shortcut methods, such as `.fadeIn()`, `.show()`, or `.slideUp()`. All of them follow a similar format, where we need to provide at least a duration, type of easing, and potentially a callback function to either perform a task when the animation has completed, or log something to the console to this effect.

All too often though, we may decide to stick with the standard values such as `slow`, `fast`, or perhaps a numerical value such as `500`:

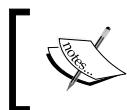
```
$("button").click(function() {
  $("p").slideToggle("slow");
});
```

There is absolutely nothing wrong with using this approach - except, it's very boring, and only using a fraction of what is possible.

Over the next few pages, we'll explore some of the tricks available to broaden our knowledge when applying effects, and realize that we don't always have to stick with the tried and tested methods. Before we explore some of these tricks, it's worth learning a little about how some of these effects are handled within the Core jQuery library.

Exploring the `animate()` method as the basis for effects

If you were asked to use a pre-configured effect such as `hide()` or `slideToggle()`, then you might be expecting to use a named function within jQuery.



It should be noted that the line numbers given in this section apply to the uncompressed version of jQuery 2.1.3, which is available from <http://code.jquery.com/jquery-2.1.3.js>.



Well, this is true, but only in part: the preconfigured functions within jQuery are all shorthand pointers to `animate()`, as shown in or around lines **6829** to **6840**. They go through a two stage process:

- The first stage is to pass one of three values to the `genFx()` method, namely `show`, `hide`, or `toggle`
- This is then passed to `animate()` to produce the final effect, at lines **6708** to **6725**

A quick look in the code shows each of the values available within jQuery, and how they are passed to `.animate()`:

```

6828 // Generate shortcuts for custom animations
6829 jQuery.each({
6830   slideDown: genFx("show"),
6831   slideUp: genFx("hide"),
6832   slideToggle: genFx("toggle"),
6833   fadeIn: { opacity: "show" },
6834   fadeOut: { opacity: "hide" },
6835   fadeToggle: { opacity: "toggle" }
6836 }, function( name, props ) {
6837   jQuery.fn[ name ] = function( speed, easing, callback ) {
6838     return this.animate( props, speed, easing, callback );
6839   };
6840 });

```

We covered the use of `animate()` in some detail back in *Chapter 6, Animating with jQuery*. It's worth touching on the following few key points about using `animate()` within our code:

- Only properties that take numeric values are supported, although there are some exceptions. Some values such as `backgroundColor`, can't be animated without a plugin (jQuery Color – <https://github.com/jquery/jquery-color>, or jQuery UI – <http://www.jqueryui.com>), along with those that can take more than one value, such as `background-position`.
- You can animate CSS properties by using any standard CSS unit where applicable – a full list can be viewed at http://www.w3schools.com/cssref/css_units.asp.
- Elements can be moved using relative values, that are prefixed with `+=` or `-=` in front of the property value. If a duration of `0` is set, the animation will immediately set the elements to their end state.
- As a shortcut, if a value of `toggle` is passed, an animation will simply reverse from where it is and animate to that end.
- All CSS properties set via a single `animate()` method will animate at the same time.

Now that we've seen how custom effects are handled within the library, let's explore creating some new effects, which combine those already available within the library.

Putting custom effects into action

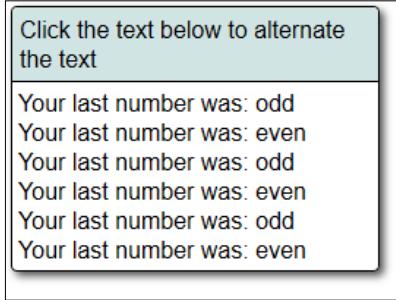
If we spent our time developing code that was restricted to using the default effects that are available within jQuery, we would quickly outgrow the limits of what can be done within it.

To prevent this from happening, it is worth spending time working out what effects we really want to use, and to see if we can't build something to replicate them from within jQuery. To prove this, we're going to delve into some examples; our first one is to produce a toggle effect based on clicking a chosen element.

Creating a clickToggle handler

The inspiration for the first of our three examples comes not from online comments, but from jQuery itself. The Core library had a `toggle` function available (as shown at <http://api.jquery.com/toggle-event/>), which was deprecated back in version 1.8, and removed in 1.9.

We're going to explore how we can add similar functionality, using a mini-plugin, the idea being that one of two functions will be run, depending on the state of a value set in our plugin:



Let's take a look and see what is required:

1. We'll start by extracting the relevant files from the code download for this book. For this demo, we'll need the `clicktoggleg.css`, `jquery.min.js`, and `clicktoggleg.html` files. Place the CSS file in the `css` subfolder, jQuery library in the `js` subfolder, and the markup file at the root of the project area.
2. In a new file, we need to create our `clicktoggleg()` event handler, so go ahead and add the following code, saving it as `clicktoggleg.js`:

```
$.fn.clicktoggleg = function(a, b) {
    return this.each(function() {
        var clicked = false;
        $(this).on("click", function() {
            if (clicked) {
                clicked = false;
                return b.apply(this, arguments);
            }
            clicked = true;
            return a.apply(this, arguments);
        });
    });
};
```



The `apply()` function is used to call the context for a function – for more details, see http://api.jquery.com/Types/#Context_2C_Call_and_Apply.

3. Immediately below the `clicktoggle` event handler, add the following functions:

```
function odd() {  
    $("#mydiv").append("Your last number was: odd<br>");  
}  
  
function even() {  
    $("#mydiv").append("Your last number was: even<br>");  
}  
  
$(document).ready(function() {  
    $("#mydiv").clicktoggle(even, odd);  
});
```

The first two look after adding the appropriate response on screen, with the third firing off the event handler when text has been clicked.

4. If all is well, we should see something similar to the screenshot shown at the start of the exercise, where we can see that the text has been clicked a few times.



A number of people have produced similar versions of this code - see <https://gist.github.com/gerbenvandijk/7542958> for one example; this version uses `data-` tags and combines the handling functions into one call.

Okay, let's move on and take a look at another example: in this one, we're going to create a slide-fade toggle effect. This will use similar principles to the previous example, where we check the state of the element. This time, we'll use the `:visible` pseudo-selector to confirm which callback message should be rendered on screen.



As an idea, why not try combining this plugin with the Toggles plugin available at <http://simontabor.com/labs/toggles/>? This could be used to produce some nice on/off buttons. We can then fire off events that are handled by the `clickToggle` plugin created in this example.

Sliding content with a slide-fade Toggle

In our previous example, our effect appeared very abruptly on screen – it was either one or the other statement, but nothing in between!

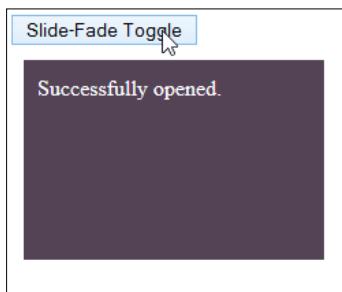
From a visual effect, this isn't always ideal; it gives a softer impression if we can make the transition smoother. Enter the Slide-Fade Toggle plugin. Let's take a look and see how to create it:

1. We'll start, as always, by extracting the relevant files that we need from the code download that accompanies this book. For this demo, we'll need the usual `jquery.min.js`, along with `slidefade.css` and `slidefade.html`. The JavaScript files need to be dropped into the `js` subfolder, the style sheet into the `css` subfolder, and the HTML markup file at the root of our project area.
2. In a new file, let's go ahead and create the `slideFadeToggle` effect. Add the following lines, saving it as `slidefade.js` in the `js` subfolder:

```
jQuery.fn.slideFadeToggle = function(speed, easing,
callback) {
    return this.animate({opacity: 'toggle', height: 'toggle'},
speed, easing, callback);
};

$(document).ready(function() {
    $("#sfbutton").on("click", function() {
        $(this).next().slideFadeToggle('slow', function() {
            var $this = $(this);
            if ($this.is(':visible')) {
                $this.text('Successfully opened.');
            } else {
                $this.text('Successfully closed.');
            }
        });
    });
});
```

3. If all is well, then when we preview the results in a browser, we should see the dark grey square fade as it slides up, once we click on the button. This is shown in the following images:



The code creates a nice alert effect – it could be used to display a suitable message to visitors within your site as it slides into view. We've based our plugin on toggling between two states. If your preference is to simply use the equivalent of the `fadeIn()` or `fadeOut()` states on their own, then we could easily use either of these functions, as appropriate:

```
$fn.slideFadeIn = function(speed, easing, callback) {  
    return this.animate({opacity: 'show', height: 'show'},  
    speed, easing, callback);  
};  
  
$fn.slideFadeOut = function(speed, easing, callback) {  
    return this.animate({opacity: 'hide', height: 'hide'},  
    speed, easing, callback);  
};
```

Okay, let's move on. We've created some custom effects, but it still feels like it's missing something. Ah yes – I know what: how about easing from one state to another? (And yes, pun absolutely intended!)

Instead of simply setting slow, fast, normal, or even a numeric value to control the duration of the effect, we can also add an easing capability that gives the effect some much needed action. Let's delve in and see what is involved.

Applying custom easing functions to effects

If someone mentions the word "easing" to you, I'll bet one of two things will happen:

- You will most likely think that you'll need to use jQuery UI, which has the potential to add a fairly significant chunk of code to the page
- You'll run away, at the thought of having to work out some horrendous math!

The irony here though, is that the answer to both could be yes and no (at least to the first part of the second comment). Hold on – how come?

The reason for this is that you most certainly don't need jQuery UI to provide special easing functions. Granted, if you are already using it, then it would make sense to use the effects contained within. While you might have to work out some maths, this would only be necessary if you really want to get stuck into complex formulae, which isn't always necessary. Intrigued? Let me explain more.

Adding an easing to code need not be any more than a simple function that uses any one of five different values, as shown in the following table:

Value	Purpose
x	null Note that although x is always included, it is nearly always set as a null value
t	Time elapsed.
b	Initial value
c	Amount of change
d	Duration

In the right combination, they can be used to produce an easing, such as the `easeOutCirc` effect, available within jQuery UI:

```
$ .easing.easeOutCirc= function (x, t, b, c, d) {
    return c * Math.sqrt(1 - (t=t/d-1)*t) + b;
}
```

Taking it further, we can always work out our own custom easing functions. A good example is outlined at <http://tumblr.ximi.io/post/9587655506/custom-easing-function-in-jquery>, along with comments indicating what needs to happen to make it work in jQuery. As an alternative, you can also try <http://gizma.com/easing/>, which lists a number of examples of similar effects.

I think it's time for us to get practical. Let's dive in and make use of these values to create our own easing function. We'll start with adding a predefined easing to one of our previous examples, before stripping it out and replacing it with a custom creation.

Adding a custom easing to our effect

We could of course use the likes of the Easing plugin which is available to download from <http://gsgd.co.uk/sandbox/jquery/easing/> or even jQuery UI itself. There is no need though. Adding a basic easing effect only requires a few lines of code.

Although the math involved may not be easy, it is a cinch to add in a specific easing value. Let's take a look at a couple of examples:

1. For this demo, we'll start by extracting the relevant files from the code download that accompanies this book. We'll need the `slidefade.html`, `slidefade.js`, `jquery.min.js`, and `slidefade.css` files. These need to be saved to the relevant folders within our project area.

2. In a copy of `slidefade.js`, we need to add our easing. Add the following code immediately at the start of the file, before the `slideFadeToggle()` function:

```
$.easing.easeOutCirc= function (x, t, b, c, d) {
    return c * Math.sqrt(1 - (t=t/d-1)*t) + b;
}
```

3. Although we've added our easing effect, we still need to tell our event handler to use it. For this, we need to modify the code as shown next:

```
$(document).ready(function() {
    $("#sfbutton").on("click", function() {
        $(this).next().slideFadeToggle(1000, 'easeOutCirc');
    });
});
```

4. Save the files as `slidefadeeasing.html`, `slidefadeeasing.css`, and `slidefadeeasing.js`, then preview the results in a browser. If all is well, we should notice a difference in how the `<div>` element collapses and fades away to nothing.

At this stage, we have a perfect basis for creating our own custom easing functions. To test this, try the following:

1. Browse to the Custom Easing Function Explorer site, which is located at <http://www.madeinflex.com/img/entries/2007/05/customeasingexplorer.html>, and then using the sliders, set the following values:

- Offset: 420
- P1: 900
- P2: -144
- P3: 660
- P4: 686
- P5: 868

2. This will produce the following equation function:

```
function(t:Number, b:Number, c:Number, d:Number):Number {
    var ts:Number=(t/=d)*t;
    var tc:Number=ts*t;
    return b+c*(21.33482142857142*tc*ts +
    - 66.94196428571428*ts*ts + 75.26785714285714*tc +
    - 34.01785714285714*ts + 5.357142857142857*t);
}
```

3. As it stands, our equation won't work when used in our code; we need to edit it. Remove all instances of :Number, then add an x before the t in the parameters. The code will look like the following when edited – I've assigned an easing name to it:

```
$.easing.alexCustom = function(x, t, b, c, d) {
    var ts=(t/=d)*t;
    var tc=ts*t;
    return b+c*(21.33482142857142*tc*ts +
    - 66.94196428571428*ts*ts + 75.26785714285714*tc +
    - 34.01785714285714*ts + 5.357142857142857*t);
}
```

4. Drop this into `slidefade.js`, then amend the easing name used in the `document.ready()` block, and run the code. If all is well, our new custom easing will be used when animating the `<div>` element.

This opens up lots of possibilities. It is feasible to write the functions we've just generated manually, but it takes a lot of effort. The best result is to use an easing function generator to produce the results for us.

Now, we can continue to work with functions such as the two we've examined here, but this seems like a tough nut to have to crack each time we want to provide some variety when animating elements! We could equally be lazy, and simply import effects from jQuery UI, but that also brings across a lot of redundant baggage; jQuery should be about providing a light touch approach!

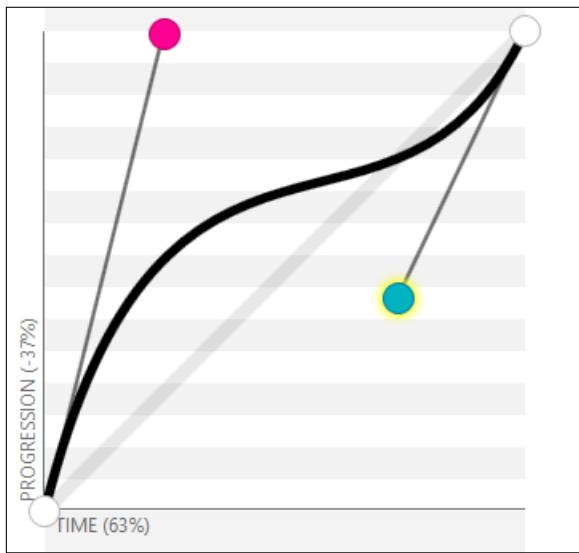
Instead, we can use a far easier option. While many might initially be scared of using Bezier curves, some kind souls have already done most of the heavy lifting for us, which makes it a breeze to use when creating effects.

Using Bezier curves in effects

A question – hands up if you can work out what Renault and Citroen have in common, apart from being two rival car manufacturers? The answer is the subject of our next topic – Bezier curves!

Yes, it may be hard to believe, but Bezier curves were used to design car bodies at Renault back in 1962, although Citroen beat them to it, using them as early as 1959.

However, I digress – we're here to look at using Bezier curves with jQuery, such as the next example:



You can view this example at <http://cubic-bezier.com/#.25,.99,.73,.44>.



These are not supported by default; an attempt was made to incorporate support for them, which wasn't successful. Instead, the easiest way to include them is to use the Bez plugin, which is available from <https://github.com/rdallasgray/bez>. To see how easy it is to use, let's take a look at it in action.

Adding Bezier curve support

There are a number of online sites that show off examples of easing functions; my personal favorites are <http://easings.net/> and <http://www.cubic-bezier.com>.

The former, by Andrey Sitnik, is one we visited back in *Chapter 6, Animating jQuery*. This provides working examples of all the easings available with jQuery. If we click on one, we can see various ways they can either be created or used within jQuery.

The easiest way to provide support is using the aforementioned Bez plugin. I think it's time for a short demo now:

1. For this demo, we'll start by extracting the relevant files from a copy of the code download that accompanies this book. We'll need the `blindtoggle.html`, `jquery.min.css`, `blindtoggle.css`, and `jquery.bez.min.js` files. These need to be stored in the relevant subfolders of our project area.
2. In a new file, let's go ahead and create the jQuery effect. In this instance, add the following to a new file, saving it as `blindtoggle.js` within the `js` subfolder of our project area:

```
jQuery.fn.blindToggle = function(speed, easing, callback) {
    var h = this.height() + parseInt(this.css('paddingTop')) +
        parseInt(this.css('paddingBottom'));
    return this.animate({
        marginTop: parseInt(this.css('marginTop')) < 0 ? 0 : -h,
        speed, easing, callback
    );
};

$(document).ready(function() {
    var $box = $('#box').wrap('<div id="box-outer"></div>');
    $('#blind').click(function() {
        $box.blindToggle('slow', $.bez([.25,.99,.73,.44]));
    });
});
```

3. If we preview the results in a browser, we can see the text first scroll up, followed quickly by the brown background, as seen in the next image:



It seems like a fair bit of code, but the real key to this demo lies in the following line:

```
$box.blindToggle('slow', $.bez([.25,.99,.73,.44]));
```

We're using the `$.bez` plugin to create our easing functions from cubic-bezier values. The main reason for this is to avoid the need to provide both CSS3 and jQuery based cubic-bezier functions; the two are not mutually compatible. The plugin gets around this by allowing us to provide easing functions as cubic-bezier values, to match those that can be used in style sheets.

Adding cubic-bezier support to our code opens up a world of possibilities. To get you started, following are some links as inspiration:

- Want to replace the standard jQuery effects such as `easeOutCubic`? No problem - <http://rapiddg.com/blog/css3-transiton-extras-jquery-easing-custom-bezier-curves> has a list of cubic-bezier values that will provide the equivalent functionality using CSS.
- If you happen to work with CSS preprocessors such as Less, then Kirk Strobeck has a list of easing functions for Less, which is available at <https://github.com/kirkstrobeck/bootstrap/blob/master/less/easing.less>.
- We talked briefly about the tool available at <http://www.cubic-bezier.com>, for working out the co-ordinate values. You can read about the inspiration behind this awesome tool, from the creator Lea Verou at <http://lea.verou.me/2011/09/a-better-tool-for-cubic-bezier-easing/>. An alternative tool is also available at <http://matthewlein.com/ceaser/>, although this is not so easy to use, and is geared more towards CSS values.

It's worth spending time getting familiar with using cubic-bezier values. It's a cinch to provide them, so it's over to you to create some really cool effects!

Using pure CSS as an alternative

When developing with jQuery, it's all too easy to fall into the trap of thinking that the effects must be provided by jQuery. It's a perfectly understandable mistake to make.

The key to becoming a more rounded developer is to understand the impact of using jQuery to provide such an effect.

On older browsers, we may not have had a choice. However, on newer browsers, we do. Instead of simply using an effect such as `slideDown()`, consider whether you can achieve the same (or very similar) effect using CSS. For example, how about trying the following as an alternative to `slideDown()`:

```
.slider { transition: height 2s linear; height: 100px;  
background: red; }  
.slider.down { height: 500px; }
```

We can then shift our focus to simply changing the assigned CSS class, thus:

```
$('.toggler').click(function() {
    $('.slider').toggleClass('down');
});
```

Ah, but – this is a book about mastering jQuery, right? And why would we want to avoid using jQuery code? Well – to quote Polonius from Shakespeare's *Hamlet* - "... Though this be madness, yet there is method in't". Or, to put it another way, there is a very sensible reason for following this principle.

jQuery is an inherently heavy library, weighing at 82 KB for a default minified copy of version 2.1.3. Granted, work is being done to remove redundant functionality, and yes, we can always remove elements we don't need.

But, jQuery is resource hungry; this puts an unnecessary burden on your site. Instead, it's far more sensible to use functionality such as `toggleClass()` – as we have here – to switch classes. We can then maintain separation with CSS classes being stored in the style sheet.

It all comes down to your requirements. If, for example, you only need to produce a couple of effects, then there is little point in pulling in jQuery for this job. Instead, we can use CSS to create these effects, and leave jQuery for where it will add most value in providing the heavy lifting within the site itself.



To prove a point, have a look at the `replacejquery.html` demo in the code download that accompanies this book. You will need to extract the `replacejquery.css` file too, to get it to work. This code creates a very basic, but functional slider effect. Look carefully, and you should not see any jQuery in sight...!

Now, don't get me wrong. There may be some instances where jQuery is a must (if for example supporting an older browser), or circumstances dictate that a neater option requires use of the library (we can't chain when using pure CSS). In these cases, we have to accept the extra burden.

To prove though that this should be the exception rather than the rule, following are some examples to entice you:

- Take a look at the well-known library `animate.css` by Dan Eden (available at <http://daneden.github.io/animate.css/>). This contains lots of CSS-only animations that can be imported into your code. If you do need to use jQuery, then the Animo jquery plugin at <http://labs.bigroomstudios.com/libraries/animo-js> is worth a look – this uses the `animate.css` library.

- Have a look at <http://rapiddg.com/blog/css3-transiton-extras-jquery-easing-custom-bezier-curves>. In the table about half way down, is a list of Bezier curve equivalents for most (if not all) of the easing effects available when using jQuery. The trick here is to not use the extra functions that we've created in previous examples, but to simply use `animate()` and the `Bez` plugin. The latter will be cached, helping to reduce the load on the server too!
- A simple, but effective example of using CSS3 to provide a simple image fade-in is available at <http://cssnerd.com/2012/04/03/jquery-like-pure-css3-image-fade-in/>. The fade transition could use a slightly longer period, but it shows the effect well.

The key message here is that it isn't always necessary to use jQuery – part of becoming a better developer is to work out when we should and should not resort to using a sledge hammer to crack that nut!

Okay, time to crack on (sorry, pun intended). Let's take a quick look at adding callbacks, and how with a change of mindset, we can replace this with an improved alternative that makes for easier use within jQuery.

Adding callbacks to our effects

Okay, so we've created our effect, and set it to run. What if we wanted to be alerted when it completes, or even if it fails? Easy! We can provide a callback, as long as we pass a function (with or without parameters). Then we can ask jQuery to perform an action once the effect is completed, as shown in the following example:

```
$ (document) .ready(function() {
    $("#myButton") .on("click", function () {
        $('#section') .hide(2000, 'swing', function() {
            $(this) .html("Animation Completed");
        });
    });
});
```

It's a perfectly workable way of being notified, and a breeze to implement. But it's not without its shortcomings. Two of the principal ones are maintaining control over when and how the callback is executed, and only being able to run one callback.

Thankfully, we are not obliged to use standard callbacks, as jQuery's Deferreds comes to the rescue. We touched on using it back in *Chapter 5, Integrating AJAX*. The beauty about Deferreds and Promises is that they can be applied to any jQuery functionality; events are particularly suited for this purpose. Let's take a look at how we can make use of this functionality, within the context of effects.

Controlling content with jQuery's Promises

Promises, promises – how many times have I heard that phrase, I wonder?

Unlike in real life, when promises made are often broken, we can always guarantee that Promises made in jQuery will be satisfied at some point. Granted, the answer may not always be positive one, but yes, there will at least be a response to a Promise.

A question though, I hear you ask – why, if most events already have callback options built in, do we need to use jQuery's `.promise()`?

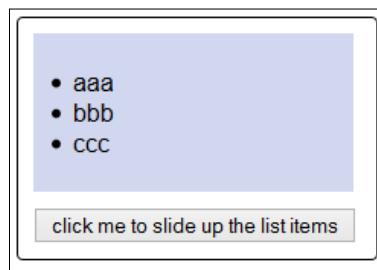
The simple answer is that we have far more control over constructing and reading Promises. For example, we can set a single callback that can be applied to multiple Promises; we can even set a Promise to only fire once, if needed! The beauty though is that using Promises makes it easier to read the code, and chain multiple methods together:

```
var myEvent = function() {
    return $(selector).fadeIn('fast').promise();
};

$.when( myEvent() ).done( function() {
    console.log( 'Task completed.' );
});
```

We can even hive off the main effect into a separate function, then chain that function to the Promise to determine how it should be handled within our code.

To see how easy it is to combine the two, let's take a moment to consider the following simple example, which uses the `slideUp()` effect in jQuery:



1. We'll start by extracting the `promises.html`, `promises.css`, and `jquery.min.js` files. Go ahead and store these in the relevant folders within our project area.

2. In a new file, add the following code— this contains a click handler for the button in our markup file, that will first slide up the `` items, then display a notice on screen when this is completed.

```
$(document).ready(function() {
    $('#btn').on("click", function() {
        $.when($('li').slideUp(500)).then(function() {
            $("p").text("Finished!");
        });
    });
});
```

3. Try running the demo in a browser. If all is well, we should see the three list items roll up when clicking on the button on screen, as shown in the screenshot at the start of this section.

This simple demo illustrates perfectly how we can use Promises to make our code more readable. Sorry to disappoint you if you were expecting more! The key here though is not necessarily about the *technical capability* of providing a callback, but the *flexibility* and *readability* gained from using Promises.



It is worth noting in this example that we are using the jQuery object's `promise()` method in this instance – we should ideally use a different object as the basis for the Promise.



To really see how Promises can be used, take a look at <http://jsfiddle.net/6sKRC/>, which shows a working example in a JSFiddle. This extends the `slideUp()` method to remove the elements in their entirety, once the animation has been completed.

It should be noted that although this shows a great way to extend this effect, the code itself could benefit from some tweaking to make it more readable. For example, `this.slideUp(duration).promise()` can easily be separated into a variable, which would make that line shorter and easier to read!



If you would like to learn more about using jQuery's Promises and Deferreds, then there are plenty of articles online on both subjects. Two that may be of interest can be found at <http://code.tutsplus.com/tutorials/wrangle-async-tasks-with-jquery-promises--net-24135> and <http://tutorials.jenkov.com/jquery/deferred-objects.html>. It's definitely worth taking time to get your head around the subject, if you've not used `promises()` before!



We're coming close to the end of this chapter, but before we round it up, there is one more important topic to cover. We've considered the benefits of using CSS in some form or other, rather than just relying on jQuery. If circumstances dictate that the latter must be used, then we should at least consider managing the queues to gain the most benefit from using effects. Let's take a moment to explore this in more detail.

Creating and managing the effect queue

Queues, queues – who likes queuing, I wonder?

Although not all of us like to queue for things, such as for getting lunch or visiting a bank, queuing is critical to the success of running animations. It matters not one jot if we're using `.slideUp()`, `.animate()` or even `.hide()` – if we chain too many animations, we will hit a point where animations won't run.

To release the animation, we need to explicitly call `.dequeue()`, as the methods come in pairs. Consider the following example for a moment, taken from <http://cdmckay.org/blog/2010/06/22/how-to-use-custom-jquery-animation-queues/>:

Imagine you're making a game and you want to have an object start at `top: 100px`, then float upwards for 2000 milliseconds. Furthermore, you would like the said object to stay completely opaque for 1000 milliseconds before slowly becoming completely transparent over the remaining 1000 milliseconds:

Time (in ms)	Top	Opacity
0	100px	1.0
500	90px	1.0
1000	80px	1.0
1500	70px	0.5
2000	60px	0.0

At first glance, it appears that the `animate` command could take care of this, as can be seen in the following code:

```
$("#object").animate({opacity: 0, top: "-=40"}, {duration: 2000});
```

Unfortunately, this code will fade the object out over 2000 ms, instead of waiting 1000 ms then fading out over the remaining 1000 ms. Delay can't help either, because it would delay the upward floating as well. At this point, we can either fiddle with timeouts or, you guessed it, use queues.

With this in mind, following is what the code would look like, altered to use `.queue()` and `.dequeue()`:

```
$("#object")
  .delay(1000, "fader")
  .queue("fader", function(next) {
    $(this).animate({opacity: 0},
      {duration: 1000, queue: false});
    next();
  })
  .dequeue("fader")
  .animate({top: "-=40"}, {duration: 2000})
```

In this example, we have two queues: the `fx` queue and the `fader` queue. First off, we setup the `fader` queue. Since we want to wait 1000 ms before fading, we use the `delay` command with 1000 ms.

Next, we queue up an animation that fades the object out over 1000 ms. Pay close attention to the `queue: false` option we set in the `animate` command. This is to ensure the animation doesn't use the default `fx` queue. Finally, we unleash the queue using `dequeue` and immediately follow it with a regular `fx`, using the `animate` call to move the top of the object up 40 pixels.

We could even turn the use of `.queue()` and `.dequeue()` into a plugin. Given that both need to be used, it would make sense to turn it into something that is easier to read in code. Consider the next example:

```
$fn.pause = function( delay ) {
  return this.queue(function() {
    var elem = this;
    setTimeout(function() {
      return $( elem ).dequeue();
    }, delay );
  });
};

$(".box").animate({height: 20}, "slow").pause( 1000 ).slideUp();
```

In the previous example, we first animate the change in height to `.box` before pausing and then sliding up the `.box` element.

The key point to note is that `queue()` and `dequeue()` are based around the `fx` object in jQuery. As this is already set by default, there is no need to specify it within our plugin.



If you're unsure about the uses of `queue()` and `dequeue()`, then it's worth taking a look at <http://learn.jquery.com/effects/uses-of-queue-and-dequeue/>, which outlines some useful case examples.

Using `.queue()` and its counterpart `.dequeue()` provides a graceful means of controlling animations. Its use is arguably more suited to multiple, complex animations, particularly where animation timelines need to be implemented. If we're only using a small number of simple animations though, then the weight of an extra plugin may not be necessary. Instead, we can simply add `.stop()` to provide a similar effect. Refer to the following:

```
$(selector).stop(true,true).animate({...}, function(){...});
```

It may not be quite as graceful, but using `.stop()` does improve the look of your animations!

Summary

Wow, we've covered a lot over the last few pages. It has certainly been intense! Let's take a breather, and recap what we've learnt.

We kicked off with a revisit on basic effects, as a reminder of what we can use in jQuery, before exploring the key differences between standard animations and effects. We then moved onto creating custom effects, with a look at the basis for all effects, before creating two examples of custom effects in code.

We then turned our focus to adding custom easings, and explored how those we saw earlier in the book can equally be applied to jQuery effects. We worked our way through an example in the form of adding Bezier curve-based easing support, before exploring how we can achieve similar effects using just CSS. We then briefly covered adding callbacks to our effects, and then explored how we can better control the callbacks by using jQuery's Deferreds / Promises options as an alternative to standard callbacks.

We then rounded up the chapter with a look at managing the effects queue. This was a good opportunity to explore the benefits of careful queue management, so that we can avoid any confusion or unexpected results when using effects within jQuery.

Moving swiftly on, it's time for some real fun! Over the next couple of chapters, we're going to explore two topics that you might not immediately associate with jQuery; we'll start with exploring the Page Visibility API, where you'll see that writing lots of complex code isn't necessarily a good thing.

9

Using the Web Performance APIs

How many times have you had a browser session running with multiple tabs? As a developer, I would expect that to almost be the norm, right?

Now, what if when you switched tabs, content was still playing on the original tab? It's really irritating, right? Sure, we could stop it, but hey, we're busy people with more important things to do...!

Thankfully, this is no longer an issue – in the age of mobile, where conservation of resources is ever more important, we can employ a few tricks to help curb our use. This chapter will introduce you to using the Page Visibility API, and show you how, with some simple changes, you can dramatically reduce the resources used by your site. Over the next few pages, we will cover the following topics:

- Introducing the Page Visibility and requestAnimationFrame APIs
- Detecting and adding support, using jQuery
- Controlling activity using the API
- Incorporating support into practical uses

Ready to make a start? Good! Let's get going...

An introduction to the Page Visibility API

Consider this scenario for a moment, if you will:

You're viewing a content-heavy site on an iPad, which is set to pre-render content. This is beginning to hammer the resources on the device, with the result that battery power is being drained quickly. Can you do anything about it? Well, on that site, probably not – but if it is a site you own, then yes. Welcome to the **Page Visibility API**.

The Page Visibility API is a nifty little API that detects when content in a browser tab is visible (that is, being viewed), or hidden. Why is this of interest? Simple – if a browser tab is hidden, then there is no point in playing media on the site, or running frequent polls to a service, right?

The net impact of using this API is aimed at reducing the use of resources and (consequently) saving power. After all, your visitors will not thank you if their batteries are drained as a result of visiting a media-heavy site!

Over the next few pages, we're going to visit this library in detail and see how we can use it with jQuery. Let's kick off with a look at browser support for the API.

Supporting the API

Unlike other APIs, support for this library is very good within all major browsers. As with many APIs, Page Visibility went through the usual process of requiring vendor prefixes, before reaching Recommendation stage at the end of October 2013. At present, none of the recent browsers (post IE8) require vendor prefixes in order to operate.

A typical code extract that uses the Page Visibility API looks like the following code snippet, when using plain JavaScript:

```
var hidden, state, visibilityChange;
if (typeof document.hidden !== "undefined") {
    hidden = "hidden",
    visibilityChange = "visibilitychange",
    state = "visibilityState";
}
```

We'll be looking at using jQuery later on this chapter.

It's trivial to implement it in code, so there is no excuse not to. To prove this, let's take a look at a demo in action.

Implementing the Page Visibility API

So far, we've been introduced to the Page Visibility API, and have covered the benefits of using it to pause content when it is not visible. It's worth spending a moment to see how we can implement it in our code, and how such a simple change can reap massive benefits.

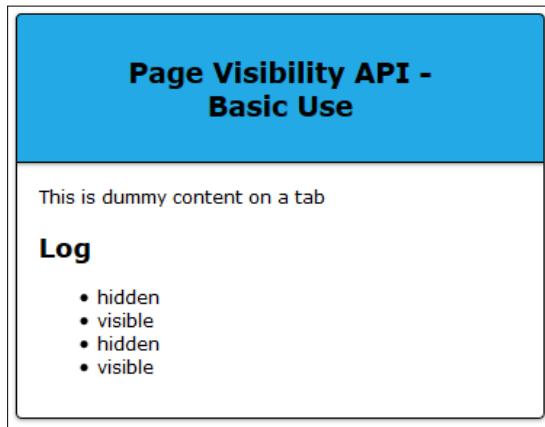
We'll begin with covering the plain JavaScript first, before looking at using jQuery later in the chapter:

1. Let's start by extracting the markup files we need from the code download that accompanies this book. For this demo, we'll need `basicuse.html` and `basicuse.css`. Save the files into the root and `css` subfolders of our project area respectively.
2. Next, in a new file add the following code:

```
function log(msg) {
    var output = document.getElementById("output");
    output.innerHTML += "<li>" + msg + "</li>";
}

window.onload = function() {
    var hidden, visibilityState, visibilityChange;
    if (typeof document.hidden !== "undefined") {
        visibilityChange = "visibilitychange";
    }
    document.addEventListener(visibilityChange, function() {
        log(document.visibilityState]);
    });
};
```

3. This is the crux of our demo, using the Page Visibility API to determine if the tab is visible or hidden. Save this in the `js` subfolder of our project area as `basicuse.js`.
4. If all is well, then when we preview the results in a browser, we should see something akin to the following screenshot – this shows the results after switching to a new tab and back again:



Breaking down the API

A quick look through the code in the previous demo should reveal two properties of note – they are `document.visibilityState` and `document.hidden`.

These form the Page Visibility API. If we look at `document.visibilityState` in more detail first, it can return any of the following four different values:

- `hidden`: Page is not visible on any screen
- `prerender`: Page is loaded off-screen, ready to be viewed by visitor
- `visible`: Page is visible
- `unloaded`: Page is about to unload (user is navigating away from current page)

We also make use of the `document.hidden` property - it is a simple Boolean property, that is set to `false` if page is visible and `true` if page is hidden.

Together with the `visibilitychange` event, we can easily be notified when the condition of a page's visibility changes. We would use something akin to the following code:

```
document.addEventListener('visibilitychange', function(event) {  
    if (!document.hidden) {  
        // The page is visible.  
    } else {  
        // The page is hidden.  
    }  
});
```

This will work for most browsers, but not all. Even though it is a minority, we still have to allow for it. To see what I mean, try running the demo in IE8 or below – it won't show anything. Showing nothing is not an option; instead, we can provide a path to degrade gracefully. So, let's take a look at how to avoid code collapsing into a heap.

Detecting support for the Page Visibility API

Although the API will work perfectly well in most modern browsers, it will fail in a limited number; IE8 is a good example. To get around this, we need to provide either a root to gracefully degrade, or use a fallback; step one to this process is to first work out if our browser supports the API.

There are different ways to do this. We could use the `Modernizr.addTest` option from Modernizr (from <http://www.modernizr.com>). Instead, we're going to use a plugin by Matthias Bynens, which contains a check for support for older browsers. The original version is available from <https://github.com/mathiasbynens/jquery-visibility>. The version included in the code download is a cut-down copy, which removes support for older browsers.



A version of this demo that uses Modernizr is available in the code download that accompanies this book. Extract and run the `usemodernizr.html` file to see how it works.



Now that we've seen how Page Visibility can be incorporated into our code, we will switch to using jQuery for this demo.

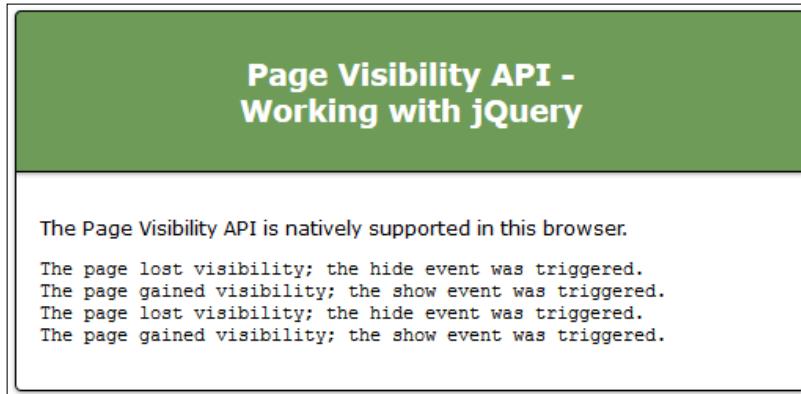
Let's start:

1. We need to start with downloading the markup and styling files from the code download that accompanies this book. Go ahead and extract copies of the following: `usingjquery.html`, `usingjquery.css`, `jquery.min.js`, and `jquery-visibility.js`. Save the CSS file to the `css` subfolder, the JS files to the `js` subfolder, and the HTML file to the root area of our project folder.
2. In a new file, add the following code – this contains the code required to check visibility, and confirm that the browser supports the API:

```
$(document).ready(function() {
    var $pre = $('pre');
    var $p = $('p')
    var supported = 'The Page Visibility API is natively
    supported in this browser.'
    var notsupported = 'The Page Visibility API is not
    natively supported in this browser.'
    $('p').first().html(
        $.support.pageVisibility ? log($p, supported) : log($p,
        notsupported)
    );
    function log(obj, text) { obj.append(text + '<br>'); }
    $(document).on({
        'show.visibility': function() {
            log($pre, 'The page gained visibility; the
            <code>show</code> event was triggered.');
        },
        'hide.visibility': function() {
            log($pre, 'The page lost visibility; the
            <code>hide</code> event was triggered.');
        }
    });
});
```

```
<code>hide</code> event was triggered.' );  
}  
});  
});  
});
```

3. Save the file as `usingjquery.js` in the `js` subfolder of our project area. If we run the demo in IE9 or above, we will see it render the changes as we switch between tabs. Refer to the following image:



4. Try changing the browser to IE8 – either by using the IE Developer Toolbar, or switching to a native copy of the browser. We also need to change the version of jQuery used, as our demo was aimed at newer browsers. Change the link to jQuery to this:

```
<script src="http://code.jquery.com/jquery-1.11.2.min.js">  
</script>
```

5. Now try refreshing the browser window. It will show that it doesn't support Page Visibility API, but equally does not crash out with unexpected errors. Refer to the next image:



With a fallback option in place, we now have two options here:

- We could just provide a path to gracefully degrade when a browser isn't supported. This is perfectly acceptable, but should be given consideration first.
- We could otherwise provide fallback support to allow for older browsers to still be used.

Let's assume we use the latter route. We can do this using any one of a number of plugins; we will use the `visibility.js` plugin, created by Andrey Sitnik, for this purpose.

Providing fallback support

Providing fallback support for any application is the bane of any developer's life. I lose count of the number of times I want to develop something that breaks new ground, yet can't. I have to provide support for older browsers that simply can't hack the new technology!

Thankfully, this is not an issue for the Page Visibility API – browser coverage is very good, although a minority number of browser versions still require some fallback support. There are a number of plugins available for this purpose – perhaps the most well-known is by Mathias Bynens, available at <https://github.com/mathiasbynens/jquery-visibility>. We saw how to use a customized version in the previous demo.

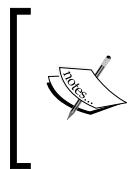
For this demo, we're going to use a similar plugin by Andrey Sitnik, which is available from <https://github.com/ai/visibilityjs>. This contains additional functionality that includes a timer to show how long your page is visible; we'll make use of this in the following demo.

Installing visibility.js

Before we begin on the demo, it's worth noting that the `visibility.js` plugin can be referenced several ways:

- We can download the original from the GitHub link at <https://github.com/ai/visibilityjs>
- It is available via Bower. To do this, you need Node and Bower installed. Once done, run the following command to download and install the plugin:
`bower install --save visibilityjs`
- It can even be referenced via a CDN link, which is currently <http://cdnjs.cloudflare.com/ajax/libs/visibility.js/1.2.1/visibility.min.js>.

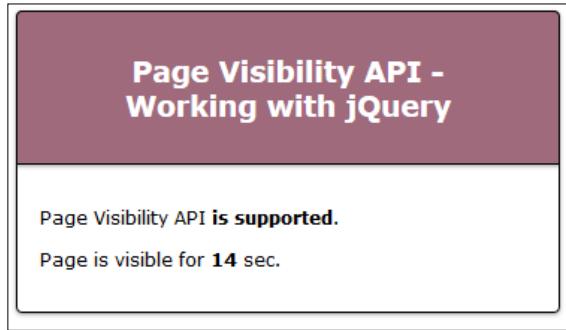
For the purposes of this demo, I'm assuming you're using the CDN version (which contains the additional timer functionality), but saved as a local copy.



Note - if you don't use this method, then you will need to download all four visibility JavaScript files at <https://github.com/ai/visibilityjs/tree/master/lib>, as these provide fallback and timer functionality that is otherwise available in the compressed CDN version.

Building the demo

Okay, now that we have our plugin in place, following is a screenshot of what we will demo:



Let's make a start:

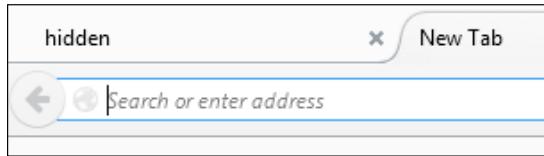
1. Extract the relevant markup files from the code download that accompanies this book. For this exercise, we'll need the `fallback.html` and `fallback.css` files. Store these in the root and `css` folders of our project area.
2. We'll also need the `visibility.min.js` plugin file - they are both in the code download file. Extract and save this to the `js` subfolder in our project area.
3. Next, add the following to a new file, saving it as `fallback.js` within the `js` subfolder of our project area:

```
$(document).ready(function() {  
    if ( Visibility.isSupported() ) {  
        $("#APIsupported").html('is supported');  
    } else {  
        $("#APIsupport").html('isn\'t supported');  
    }  
})
```

```
document.title = Visibility.state();
Visibility.change(function (e, state) {
    document.title = state;
});

var sec = 0;
Visibility.every(1000, function () {
    $("#APICounter").html(sec++);
});
});
```

4. This code contains the magic required for our demo.
5. Save the files. If we preview the results in a browser, we can expect to see something akin to the screenshot at the start of the exercise. If we switch to a different tab, as shown in the next screenshot, then the timer count is stopped temporarily, and the original tab's title is updated accordingly:



So, what happened? It's a really easy demo, but we first kicked off with a check to ensure our browser can support the API. In most cases, this isn't an issue, except for IE8 or below.

We then displayed the initial state of the window in its title area; this is updated each time we switch from the demo to a different tab and back. As a bonus, we made use of the `visibility.timer.js` plugin that comes with the main plugin, to show a count of how long our window has been visible. This of course is stopped each time we flip over to a different browser window and back again!

The great thing though, is that unlike the previous demo, the plugin will still work even if we're using IE8 or below; we might need to alter the code in our demo to ensure it is styled correctly, but this is a minor consideration.

Let's move on. Now that we understand the basics of using the Page Visibility API, I am sure you are asking the question: how can we use it in a practical context? No problem – let's take a look at some possible use cases.

Using the API in a practical context

The API can be used in a variety of different contexts. The classic is usually to help control playback of video or audio, although it can be used with other APIs such as the Battery API, to prevent content being displayed at all if power levels are too low.

Let's take a moment to delve into some practical examples, so we can see how easy it is to implement the API.

Pausing video or audio

One of the most common uses of the API is to control playback of audio or media such as videos. In our first example, we're going to use the API to play or pause a video when switching between tabs. Let's delve in and take a look.

For this demo, we'll use a couple of additional items – the Dynamic Favicons library this is available from <http://softwareas.com/dynamic-favicons/>. Although a couple of years old, it still works OK with current versions of jQuery. The videos came from the Big Buck Bunny project website, at <https://peach.blender.org>.



The videos for this demo are from the Blender Foundation, and are
(c) copyright 2008, Blender Foundation / www.bigbuckbunny.org.



Right! Let's get cracking:

1. As always, we need to start somewhere. For this demo, go ahead and extract the pausevideo demo folder from within the code download that accompanies this book.
2. Open the pausevideo.js file. This contains the code to play or pause the video, using the jquery-visibility plugin. Refer to the following code:

```
var $video = $('#videoElement');

$(document).on('show.visibility', function() {
    console.log('Page visible');
    favicon.change("img/playing.png");
    $video[0].play();
});

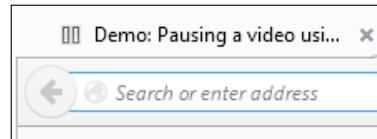
$(document).on('hide.visibility', function() {
    console.log('Page hidden');
    favicon.change("img/paused.png");
    $video[0].pause();
});
```

3. The plugin is very simple. It exposes two methods, namely `show.visibility` and `hide.visibility`. Try running the demo now. If all is well, we should see the Big Buck Bunny video play; it will pause when we switch tabs.

Following is the screenshot of the video:



4. In addition, the window's title is updated using the `favicon.js` library. It shows a pause symbol when we switch tabs, as seen in the next image:



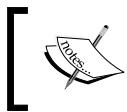
That was easy, huh? That's the beauty of the API. It is very simple, but works with a variety of different tools. Let's prove this, by incorporating support for the API into a **Content Management System (CMS)**, such as WordPress.

Adding support to a CMS

So far, we've seen how easy it is to incorporate support for the standard within static page sites – but what about CMS systems, such as WordPress, I hear you ask?

Well, the API can easily be used here too. Rather than talk about it, let's take a look and see how we can add it in. For this demo, I will use WordPress, although the principles will equally apply to other CMS systems such as Joomla. The plugin I will use is my own creation.

It should be noted that you should have a working WordPress installation available, either online or as a self-hosted version, and that you have some familiarity installing plugins.



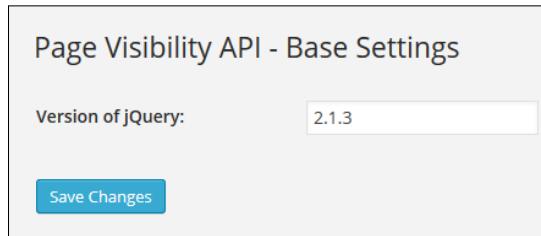
Please note – the `jquery-pva.php` plugin is *only intended for development purposes*; it needs further work before it can be used in a production environment.



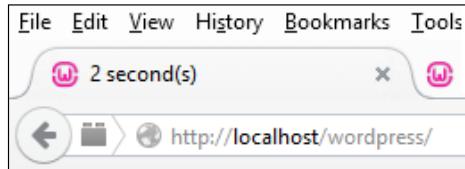
Okay, let's start:

1. We need to make changes to the `functions.php` file within a theme. For this purpose, I will assume you are using the Twenty Fourteen theme. Open `functions.php`, and then add the following code:

```
function pausevideos() {  
    wp_register_script('pausevideo', plugins_url( '/jquery-  
    pva/pausevideo.js'), array('jquery'), '1.1', true);  
    wp_enqueue_script('pausevideo');  
}  
  
add_action( 'wp_enqueue_scripts', 'pausevideos' );
```
2. From the code download that accompanies this book, find and extract the `jquery-pva` folder, then copy it to your WordPress installation; it needs to go into the `plugins` folder. Return to your WordPress installation, then activate the plugin in the usual way.
3. Next, log into your WordPress Admin area, then click on **Settings | PVA Options**, and enter the version number of jQuery that you would like to use. I will assume 2.1.3 has been chosen. Click on **Save Changes** for it to take effect. Refer to the following image:



At this point, we can begin to use the library. If we upload a video and add it to a post, it will show the time elapsed when we begin to play it; this will pause when we switch tabs:



To confirm it is working, it is worth looking in the source, using a DOM Inspector. If all is well, we should see the following links. The first link would confirm that the Page Visibility library is referenced, as shown next:

```
[+] <script src="//cdnjs.cloudflare.com/ajax/libs/visibility.js/1.2.1/visibility.min.js?ver=4.1" type="text/javascript">
```

The second link would confirm that our script is being called, as seen in the following image:

```
[+] <script src="http://localhost/wordpress/wp-content/plugins/jquery-pva/pausevideo.js?ver=1.1" type="text/javascript">
```

As we can see, the API certainly has its uses! Throughout this chapter, I've tried to keep the code relatively simple, so that it is easily picked up. It's now over to you to experiment and take it further - perhaps I can give you some ideas for inspiration?

Exploring ideas for examples

The basic principles of the Page Visibility API are simple to implement, so the level of complexity that we go to is only limited by one's imagination. During my research, I came across some ideas for inspiration – hopefully the following will give you a flavor of what is possible:

- Animations! Sometimes we can get issues with synching animations, if a tab is not active. <http://greensock.com/forums/topic/9059-cross-browser-to-detect-tab-or-window-is-active-so-animations-stay-in-sync-using-html5-visibility-api/> explores some of the tips available to help work around some of these issues.

- This next one could either freak you, or just be plain irritating – take a look at <http://blog.frankmtaylor.com/2014/03/07/page-visibility-and-speech-synthesis-how-to-make-web-pages-sound-needy/>, where the author has mixed both the Page Visibility and Speech Synthesis APIs. Be warned – he counsels against mixing the two; let us just say that this is likely to be more of a turn off! (It's included here for technical reasons only – not because we should do it.)
- A somewhat more useful technique is to use the Page Visibility API to reduce the number of checks for new emails or news feeds. The API would check to see if the tab is hidden, and reduce the frequency of requesting updates until the tab becomes active again. The developer Raymond Camden has explored the basics required to do this, so head over to his site to learn more, at <http://www.raymondcamden.com/2013/05/28/Using-the-Page-Visibility-API>.
- To really mix things up, we can instigate some useful notifications, using the Page Visibility, Web Notification, and Vibration APIs at the same time. Have a look at <http://www.binpress.com/tutorial/building-useful-notifications-with-html5-apis/163> for ideas on how to mix the three together within the site or application.

Okay, I think it's time for a change. Let's move on and take a look at another API that was created around the same time as the Page Visibility API, and works using similar principles to help reduce demand on resources.

I'm of course referring to the `requestAnimationFrame` API. Let's delve in and find out what it is, what makes it tick, and why such a simple API can be a real boon to us developers.

Introducing the `requestAnimationFrame` API

The shift to working online over the last few years has led to a massive increase in demand for performant browsers, while at the same time reducing resource consumption and battery power.

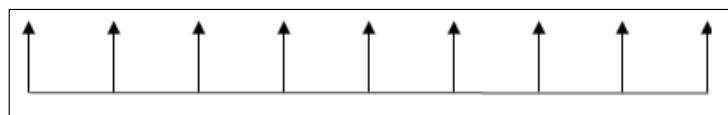
With this in mind, browser vendors and Microsoft teamed together to create three new APIs. We've already explored one, in the form of the Page Visibility API; the other that we're going to look at is `requestAnimationFrame`. All three (the third being `setImmediate`) were designed for better performance and increased power efficiency.

Exploring the concept

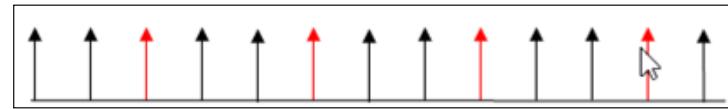
So, what is `requestAnimationFrame`? Simple – if you've spent any time creating animation using jQuery, you will no doubt have used the `setInterval` method (or even `clearInterval`), right? `requestAnimationFrame` (and `clearAnimationFrame`) were designed as drop-in replacements for each respectively.

Why should we use it? We will explore the benefits of using `requestAnimationFrame` in the next section, but first let's understand its essence.

Most animations work to a JavaScript based timer of less than 16.7ms when drawing animations, even though monitors can only display at 16.7ms (or 60Hz frequency), as indicated by the following graph:

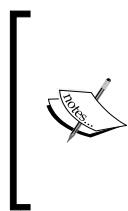


Why is this important? The key to this is that a typical `setInterval` or `setTimeout` frequency is usually around 10ms. This means that every third draw of the monitor is not seen by the viewer, as another draw will happen before the display refreshes. Refer to the next graph:



This results in a choppy display, as frames will be dropped. Battery life can be impacted by as much as 25 percent, which is a significant loss!

Browser vendors recognized this, so came up with the `requestAnimationFrame` API. This tells the application when the browser needs to update the screen, and when the browser needs a refresh. This results in a reduction in use of resources, and fewer dropped frames, as the frame rate is more consistent compared to code.



The developer Paul Irish sums it up perfectly with the following comment on his blog at <http://www.paulirish.com/2011/requestAnimationFrame-for-smart-animating/>, when he notes that this allows browsers to "optimize concurrent animations together into a single reflow and repaint cycle, leading to higher fidelity animation."

Viewing the API in action

As is nearly always the case, it is better seeing something in action, rather than reading about it. It's something about a moving demo that helps ram the concept home, at least for me!

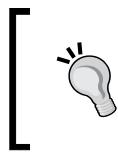
To help with this, there are two demos available on the code download that accompanies this book – the `requestAnimationFrame.html` and `cancelAnimationFrame.html` files. They contain simple examples of both APIs. We will explore more practical uses of the APIs towards the end of this chapter.

Using the requestAnimationFrame API

Although it may not be immediately apparent from the simple demos that we referenced at the end of the previous section, there are some clear benefits to using `requestAnimationFrame`, which are listed next and are worth noting:

- `requestAnimationFrame` works with the browser to combine animations together into a single repaint during a redraw transition, using the screen refresh rate to dictate when these should happen.
- Animations are paused if a browser tab is inactive or hidden, which reduces requests to refresh the screen, resulting in lower memory consumption and battery use on mobile devices.
- Animations are optimized by the browser, and not in code – the lower frame refresh rate results in a smoother, more consistent appearance, as fewer frames will be dropped.
- The API is even supported on most mobile devices too. The only platform that doesn't support it at present is Opera Mini version 8.0. The CanIUse site (<http://www.caniuse.com>) shows global usage of this as being very low at 3 percent, so this is unlikely to present too much of an issue.

It's worth noting that `cancelAnimationFrame` (as a sister API to `requestAnimationFrame`) can be used to pause animations. We can potentially use this with something like the Battery API to stop animations (or media such as videos) from kicking in, if battery power is too low.



To see the difference between `requestAnimationFrame` against `setTimeout`, then take a look at <http://jsfiddle.net/calpo/H7EEE/>. You can clearly see the difference between the two, despite the simple nature of the demo!

A key point to note though, is that there are instances where `requestAnimationFrame` doesn't always produce an improvement over using jQuery. There is a useful article by David Bushell at <http://dbushell.com/2013/01/15/re-jquery-animation-vs-css/>, which outlines this issue, and notes that `requestAnimationFrame` is best suited to being used in `<canvas>` based animations.

Creating animations based around `requestAnimationFrame` (and `cancelAnimationFrame`) is very straightforward. The developer Matt West has created a JavaScript/jQuery example on CodePen, which is available at <http://codepen.io/matt-west/full/bGdEC>. He has written a tutorial that accompanies this demo, and which is available on Team Treehouse's blog at <http://blog.teamtreehouse.com/efficient-animations-with-requestanimationframe>.

This brings us nicely onto our next subject. Now that we've seen how to manipulate the API using JavaScript, let's take a look at using similar techniques with jQuery.

Retrofitting the changes to jQuery

So far, we've covered the basics of using `requestAnimationFrame`, along with its sister API, `cancelAnimationFrame`; we've seen how to implement it using either plain JavaScript.

It's worth noting at this point though that jQuery does not have native support included. An attempt was made to add it to jQuery prior to version 1.8, but was removed due to issues with support by the major browser vendors.

Thankfully, vendor support is now much better than previously; and there are plans to add `requestAnimationFrame` support into jQuery 2.2 or 1.12. You can see the changes that need to be made as follows, along with the history:

- The commit: <https://gitcandy.com/Repository/Commit/jquery/72119e0023dcc0d9807caf6d988598b74abdc937>
- The changes to `effect.js` which can be referred from <https://github.com/jquery/jquery/blob/master/src/effects.js>
- Some of the history behind including `requestAnimationFrame` in jQuery core: <https://github.com/jquery/jquery/pull/1578>; <http://bugs.jquery.com/ticket/15147>

As a temporary measure (if you still need to support an earlier version of jQuery), you can try using Corey Frang's drop-in shim at <https://github.com/gnarf/jquery-requestAnimationFrame>, which adds support to versions of jQuery post 1.8.

If however you are feeling more adventurous, then it is easy enough to retrofit `requestAnimationFrame` support directly to a library that uses it. Let's take a moment to see what is involved in making the conversion.

Updating existing code

Making the change is relatively straightforward. The key to it is making the changes modular so that they can be swapped back out easily, once jQuery gains support for `requestAnimationFrame`.

The changes can be made if the library you are using has code references to either `setInterval` or `clearInterval`. For example, consider if we had the following code extract:

```
var interval = setInterval(doSomething, 10)
var progress = 0
function doSomething() {
    if (progress != 100) {
        // do something here
    }
    else {
        clearInterval(interval)
    }
}
```

It would be updated to the following code extract, replacing the reference to `setInterval` with `requestAnimationFrame` (and adding the equivalent replacement for `clearInterval`):

```
var requestAnimationFrame = window.requestAnimationFrame;
var cancelAnimationFrame = window.cancelAnimationFrame;

// your code here

var progress = 0;

function doSomething() {
    if (progress != 100) {
        // do something here
        var myAnimation = requestAnimationFrame(doSomething);
    } else {
        cancelAnimationFrame(myAnimation);
    }
}
```

In the previous code example, the code highlighted in bold indicates the type of changes needed to update the code. We will use this technique later on in this chapter, to retrofit support to an existing library. It will be one of two demos that we will explore, which use `requestAnimationFrame`.

Some examples of using `requestAnimationFrame`

Up until now, we've seen the theory behind using `requestAnimationFrame` and covered the typical changes that we might have to make to existing code.

It's a good starting point, but not always easy to get one's head around the concept; it's much easier to see in action! With this in mind, we're going to take a look at a couple of demos, which make use of the API. The first will retrofit support, whilst the second has been built with support already included in the code.

Creating a scrollable effect

For our first demo, we're going to take a look at updating an example of the classic scrollable UI element that can be found on hundreds of sites worldwide. We'll be using the Thumbelina plugin, available from <https://github.com/StarPlugins/thumbelina>. Although it is a couple of years old, it still works perfectly well, even with the latest version of jQuery!

In this demo, we'll replace the `setInterval` call within the plugin, to `requestAnimationFrame`. Let's start:

1. Let's begin by extracting a copy of the `thumbelina` demo folder that is in the code download that accompanies this book. If we run the `scrollable.html` file, we should see a scrollable appear with images of orchids, as in the following image:



2. The Thumbelina plugin currently uses `setInterval` to manage the time period between animations. We're going to alter it to use the new `requestAnimationFrame` instead.
3. Open `thumbelina.js`, then add the following code immediately below `$.fn.Thumbelina = function(settings) {`, which is at line 16:

```
var start = new Date().getTime(),
handle = new Object();
function loop() {
    var current = new Date().getTime(),
    delta = current - start;
    if(delta >= delay) {
        fn.call();
        start = new Date().getTime();
    }
    handle.value =
        window.requestAnimationFrame(loop);
};
handle.value = window.requestAnimationFrame(loop);
return handle;
}
```

4. Scroll down to the following line, which will be on or around line 121:
`setInterval(function() {`

5. Modify it as shown next, so it uses the new `requestInterval()` function we have just added:

```
requestInterval(function() {
    animate();
}, 1000/60);
};
```

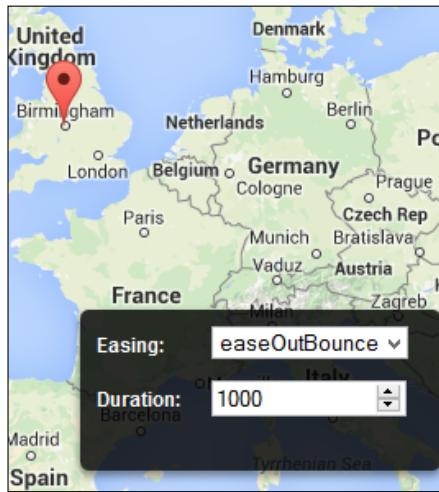
6. Save the file. If we run the demo, we should not see any visual difference; the real difference is what happens in the background.



Try running the demo in Google Chrome, then viewing the results within the Timeline. If you do a before and after, you should see a significant difference! If you are unsure how to profile the demo, then head over to <https://developer.chrome.com/devtools/docs/timeline> for full details.

Animating the Google Maps marker

Our final demo in this chapter makes use of the well-known Google Maps service, to animate moving the marker that indicates a specific location on the map:



In this example, we're going to use the demo created by Robert Gerlach, which is available from <http://robsite.net/google-maps-animated-marker-move/>. I've tweaked the code in his `markerAnimate.js` plugin file to remove the vendor prefixes, as these are no longer required.

He's produced a neat effect that gives some life to what can appear very plain content. Nonetheless, it still requires a fair amount of code! Space constraints mean we can't explore all of it in print, but we can explore some of the more important concepts:

1. Let's begin by extracting the `googlemap` demo folder from the code download that accompanies this book. This contains the styling, JavaScript libraries, and markup for our demo.
2. Run `googlemap.html` in a browser. If all is well, we should see the pointer over Birmingham, UK, where Packt Publishing's UK office is based.

Try clicking somewhere else on the map – notice how it moved across? It's taking advantage of some easing effects available in the jQuery Easing plugin, which we used back in *Chapter 6, Animating in jQuery*.

We can choose which easing effect to use by simply changing the value shown in the drop-down box in the bottom right corner. This could even include a custom animation that we craft, using the examples given in *Chapter 6, Animating in jQuery* as a basis. As long as the custom animation function is included in our code, and the appropriate name is added to the dropdown, we can use it.

The real point to note is actually in the `markeranimate.js` file. If we open it and scroll down to lines 64 - 71, we can see how `requestAnimationFrame` has been used. We use it if the browser supports the API, otherwise `setTimeout` is used, as shown in the following screenshot:

```
// use requestAnimationFrame if it exists on this browser. If not, use setTimeout with ~60 fps
if (window.requestAnimationFrame) {
    marker.AT_animationHandler = window.requestAnimationFrame(function() {animateStep(marker, startTime)});
} else {
    marker.AT_animationHandler = setTimeout(function() {animateStep(marker, startTime)}, 17);
}
} else {
```

The combination of using an easing and calling `requestAnimationFrame` makes for a cool effect, that also reduces the demand on resources – great if you have a lot of animations on your site!



To make it easier to incorporate replacements for `setInterval`, `clearInterval` (and `setTimeout` / `clearTimeout`), use the replacement functions by Joe Lambert, which are available at, <https://gist.github.com/joelambert/1002116>.

Exploring sources of inspiration

We've covered a lot over the last few pages – it can take some time to fully appreciate how `requestAnimationFrame` (and its sister, `clearAnimationFrame`) work, but with the upcoming changes to `jQuery`, it is worth spending the time to familiarize ourselves with the APIs and the benefits they bring to our development.

Before we round up this chapter, listed next are a couple of sources of inspiration that you may find useful:

- `requestAnimationFrame` is by no means limited to playing videos or music, or the like. It can even be used in developing online games! Take a look at <http://www.somethinghitme.com/2013/01/09/creating-a-canvas-platformer-tutorial-part-one/> - hopefully you'll recognize some of the classics!

- On a more serious note, and for those sites using parallax scrolling, there is likely to be room for improvement in the implementation. Krister Kari has written a detailed blog post that goes through a typical example, and outlines some of the techniques that can be used to fix the issues. You can read it at <http://kristerkari.github.io/adventures-in-webkit-land/blog/2013/08/30/fixing-a-parallax-scrolling-website-to-run-in-60-fps/>.

There are plenty more sources available – over to you to see where your imagination takes you!

Summary

Delving into new APIs is always fun. Even though they can be simplistic in nature (check out the Vibration API, for example), they can prove to be a really useful addition to anyone's toolbox. We've explored two in detail in this chapter. Let's take a moment to recap what we've covered.

We kicked off with an introduction to the Page Visibility API. We looked at browser support for the API, before implementing a basic example. We moved onto how to detect and provide fallback support, and then looked at some practical examples.

Next came a look at the requestAnimationFrame API, where we learnt about some of the similarities to the Page Visibility API. We explored the basics of how it worked, before looking at some practical uses and how to add support to jQuery itself. We then rounded up the chapter with a look at two examples; one based around converting to using the API, whilst the other had it built in from the ground up.

Moving on, in the next chapter we'll explore another key element of websites, namely images. We're going to explore how you can manipulate images using jQuery to produce some really interesting effects.

10

Manipulating Images

It is often said that images paint a thousand words – websites are no different.

We use images to illustrate a process, help reinforce a message, or apply some visual identity to what otherwise might be seen as very plain content. Images play a key part of any website; the quality of images will either make or break a site.

A small part of using jQuery to manipulate images is how we can apply filters, or manipulate the colors within images. In this chapter, we'll explore how you can use jQuery to manipulate images, before exploring a couple of real-world examples of capturing images as a basis for further manipulation. In this chapter, we'll cover the following topics:

- Applying filters using CSS and jQuery
- Using plugins to edit images
- Creating a simple signature pad using jQuery and canvas
- Capturing and manipulating webcam images

Let's start...!

Manipulating colors in images

A question – how often have you assumed that the only way to manipulate an image is to use the likes of Photoshop, or even GIMP? I'll bet it is more than once – what if I said that heavyweight applications such as these well-known applications are (in some cases) redundant, and that all you need is a text editor and a little jQuery?

At this point, you're probably wondering what we can do to manipulate images using jQuery. Fear not! There are a few tricks up our sleeve. Over the next few pages, we're going to take a look at each, and discover that while we can use what is arguably one of the most popular JavaScript libraries available to developers, it isn't always the right way to do things.

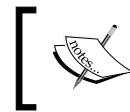
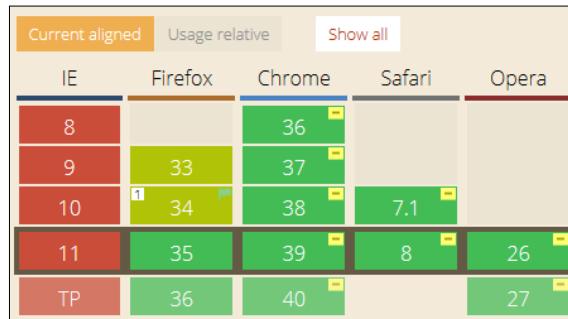
To see what I mean, let's quickly recap the methods we can use, which are:

- Using CSS3 filters, and switching them in or out using jQuery
- Using a mix of the HTML5 `<canvas>` element, jQuery, and the `getImageData` method handler to manipulate the color elements of each image, before repainting it back to the canvas

In this chapter, we'll take a look at each in turn, and explore why even though we may be able to create complex filters using jQuery, it isn't always the right answer. Hopefully, with a few tricks up our sleeve, it will make us better developers. Let's begin with a look at using simple CSS3 filters, and how we can easily incorporate their use into our jQuery code.

Adding filters using CSS3

Filter support has been available for some time, at least within the major desktop browsers, although we still need to use the `-webkit-` vendor prefix support, as we are not yet entirely prefix free:



Information about the preceding image is taken from the CanIUse website, at <http://caniuse.com/#feat=css-filters>.



The beauty about using these methods is that they are very simple to apply; we're not forced to spend hours reworking images if clients decide to change their minds! We can apply and remove the styles using jQuery with ease, which helps keep the styles separate from our markup.

Manipulating images can get very complex – in fact, to cover the math involved, we could probably fill a book in its own right! Instead, we'll begin with a simple recap of using CSS3 filters, before moving onto creating more complex filters, and finishing with a couple of demos that help capture images from two unlikely sources.

Intrigued? All will become clear towards the end of this chapter, but we will first begin with a simple exercise to reacquaint ourselves with applying CSS3 filters.

Getting ready

Before we get stuck into our exercises, I would strongly recommend using Firefox or IE for these demos; if you use Chrome, then some of the demos will show Cross-Origin errors if run locally.

A good example is the cross-platform application XAMPP (available from <http://www.apachefriends.org>), or you can try WAMPServer (for PC, from <http://www.wampserver.com/en>), or MAMP (for Mac, from <http://www.mamp.info>). I will assume that you are running the demos from within a web server.

Creating our base page

In our first demo for this chapter, we're going to start with a simple recap of using the `addClass` method to apply a specific filter to an image on the page. We'll be using the Polaroid effect, developed by the Canadian developer Nick La, and available from <http://webdesignerwall.com/demo/decorative-gallery-2/>. The `.addClass()` method is something you will almost certainly have used countless times before; we're using it here as an introduction to more complex effects later in this chapter. Let's begin:

1. Let's start by downloading and extracting the following files from the code download that accompanies this book:
 - `cssfilters.html`
 - `cssfilters.css`
 - `jquery.min.js`
 - `cssfilters.js`

2. Drop the HTML markup file into the root of our project area, and the JavaScript and CSS files into the relevant subfolders in our project area.
3. In a new file, go ahead and add the following simple block of code – this is the event handler for the button, which we will use to change the filter state:

```
$(document).ready(function(){
    $("input").on("click", function(){
        $("img").toggleClass("change-filter");
    })
});
```

4. At this stage, try previewing the results in a browser. If all is well, we should see a picture of blue flowers, set in a Polaroid effect background. Refer to the following image:



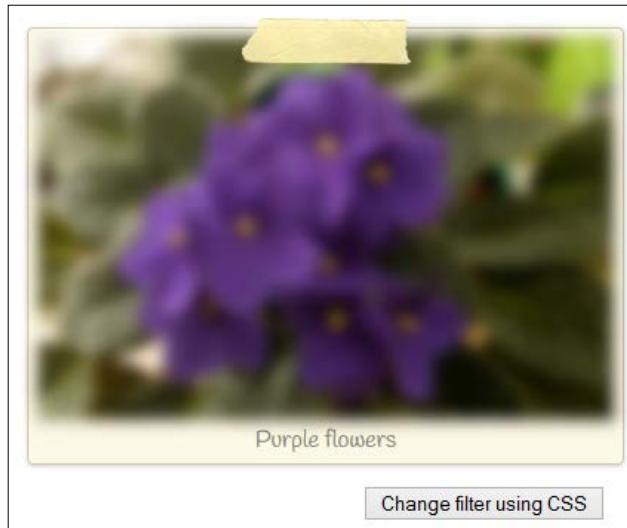
5. Take a closer look in `cssfilters.css` – near the bottom of the screen. We should see the following:

```
.change-filter {
    filter: blur(5px);
    -webkit-filter: blur(5px);
}
```

This is immediately followed by this block:

```
img { -webkit-transition: all 0.7s ease-in-out; transition: all
0.7s ease-in-out; }
```

- Now click on the **Change filter using CSS** button. If all is well, our image should gradually become blurred, as shown in the next image:



A nice simple demo – nothing too taxing at this stage, given some of the more complex topics we've covered in this book till now!

A tip – if you find that the filter doesn't display in some versions of Firefox, then check the `layout.css.filters.enabled` property in `about:config`. It is not enabled by default in version 34 or earlier; this changed from version 35:



Preference Name	Status	Type	Value
<code>layout.css.filters.enabled</code>	user set	boolean	true

The key to this demo is of course the use of the `.addClass()` method handler. We're simply applying a new, preset class to the image, when clicking the button. The beauty here though is that we have access to a number of quick and easy filters that can be used, and which can reduce (or even eliminate) the use of PhotoShop or GIMP. To see how really easy it is to swap over, let's make that change now, and switch to using the brightness filter.

Changing the brightness level

This next demo is a quick and easy change to the `cssfilters.css` file we've just been working on. Following is a screenshot of what we will produce:



Make sure you have this file available before continuing with the steps listed next:

1. In `cssfilters.css`, look for and amend the `.change-filter` rule as shown:

```
.change-filter { filter: brightness(170%); -webkit-filter: brightness(170%); }
```
2. Click on **Change filter using CSS** now. If all is well, we should find that the image has become brighter.

Again – nothing taxing here; hopefully this is a good point for a breather, after some of what we've covered in this book! There are a good handful of CSS3 filters we can use; space constraints means we can't cover them all here, but we can at least look at one more filter. The other filters available for use are outlined immediately following this next exercise.

Adding a sepia filter to our image

As before, we need to revert back to changing `cssfilters.css`, so make sure you have this ready for use. Let's take a look at what we need to do:

1. Revert back to `cssfilters.css`, then alter this line as shown:

```
.change-filter { filter: sepia(100%); -webkit-filter: sepia(100%); }
```

- Click on **Change filter using CSS** now. If all is well, we should find that the image now has a sepia filter applied, as shown in this screenshot:



This is what I love about using CSS3 filters – despite what some purists may say, it is not always necessary to revert back to using a graphics package; a simple change of a value in CSS is all that is required.

We could manually change that value if needed, but we now have the flexibility to programmatically change it too, with little impact on performance. This last point is important – as we will see later in this chapter. Creating complex filters to manipulate images using jQuery is a resource hungry process, so it's not one to be done too frequently.

Exploring other filters

Before we move on and take a look at a different way of manipulating images, the following table gives you a flavor of the different filters available; all of them can be set using jQuery as outlined in our previous exercises:

Name of filter	Example of how to use it
contrast()	.change-filter { filter: contrast(170%); -webkit-filter: contrast(170%); }
hue-rotate()	.change-filter { filter: hue-rotate(50deg); -webkit-filter: hue-rotate(50deg); }
grayscale()	.change-filter { filter: grayscale(100%); -webkit-filter: grayscale(100%); }

Name of filter	Example of how to use it
invert()	.change-filter { filter: invert(100%); -webkit-filter: invert(100%); }
Saturate()	.change-filter { filter: saturate(50%); -webkit-filter: saturate(50%); }

To see examples of these in action, it is worth taking a look online - there are plenty of examples available. As a starting point, have a look at the article by Johnny Simpson at <http://www.inserthtml.com/2012/06/css-filters/>; although it is a couple of years old, and some of the settings have been tweaked since then, it still gives a useful flavor of what is possible with CSS3 filters.

Let's change track for a moment – while we can use simple CSS3 filters to manipulate aspects such as contrast and brightness, we can use an alternative method: background blending.

Blending images using CSS3

There may be instances where we prefer not to manipulate the image directly, but alter a background image instead. Similar effects are easy to achieve in static images within PhotoShop, but are less common on the Internet.

Thankfully, we can achieve the same effect using the `background-blend` mode within CSS – this has the effect of allowing us to merge two images together. Using `background-blend` mode (for which browser support is good within desktop browsers) removes the need to manually edit each photo, so if any are changed, the same effect can easily be applied to their replacements.

In the same vein as those filters we've already examined, we would apply the filters within CSS. We can then switch them on or off using jQuery at will. I won't revisit the jQuery code that would be required, as we've already seen it earlier in the chapter; suffice to say that we would apply the `background-blend` mode, using an example such as the following:

```
<style>
.blend { width: 389px; height: 259px; background:#de6e3d
    url("img/flowers.jpg") no-repeat center center; }
.blend.overlay { background-blend-mode: overlay; }
</style>
</head>
```

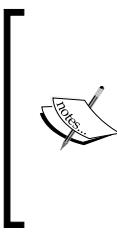
In this instance, we've used the *overlay* filter. This complex filter multiplies the colors, depending on the backdrop color value. It has the net effect of making lighter colors go lighter, and darker colors go darker, as shown in the next screenshot:



There are two examples of this blend mode in the code download that accompanies this book – look for the *overlay.html* and *multiply.html* files.



There are a good number of filter options available, such as multiply, lighten, dodge, and color burn – these are intended to produce similar effects to those used in PhotoShop, but without the need for expensive applications. All the filters follow a similar format. It is worth searching Google for examples of how filters appear, such as those shown at <http://www.webdesignerdepot.com/2014/07/15-css-blend-modes-that-will-supercharge-your-images/>.



If you would like to learn more, then head over to Mozilla's Developer site at <https://developer.mozilla.org/en-US/docs/Web/CSS/background-blend-mode>. For a really useful example of this filter (and a source of inspiration for combining it with jQuery), check out the 2016 American Presidential Candidates demo at <http://codepen.io/bennettfeely/pen/rxoAc>.



Okay, time to get really stuck into some jQuery, methinks! Let's switch to using plugins, and see some of the effects we can achieve with what is available for use. We'll start with a look at using CamanJS as our example, following it with a more in-depth exploration of creating filters manually, and see why it's not always the best way to achieve the desired effect!

Applying filters with CamanJS

So far, we've applied filters using CSS3. This is perfect for lightweight solutions, but there may be occasions where we need to do more, and CSS3 won't suffice.

Enter jQuery! Over the next few pages, we'll take a brief look at applying filters using CamanJS as our example jQuery plugin. We'll then move on and see how easy (or complex) it is to create the same effects manually, without needing to rely on a third-party plugin.

Introducing CamanJS as a plugin

CamanJS is one of the several plugins available for jQuery, which allows us to apply any number of filters; we can choose from either the preset ones that come with the library, or create our own combinations.

The plugin is available from <http://camanjs.com/>, and can be downloaded from GitHub at <https://github.com/meltingice/CamanJS>. Else, we can use NodeJS or Bower to install the library. The plugin is also available via CDN at <http://www.cdnjs.com> – search for CamanJS to get the latest URL to use in your project.

It is worth noting that filters can be applied using one of two methods – the first is as a HTML data- attribute:

```

```

The second method is using jQuery, as we will see in the next demo; we'll be using this method throughout our examples. With this in mind, let's get cracking, and take a look at using CamanJS to apply filters, as shown in our next demo.

Building a simple demo

In this demo, we'll be using the CamanJS library to apply any one of the three filters to our flowers image that we've been using throughout this chapter.

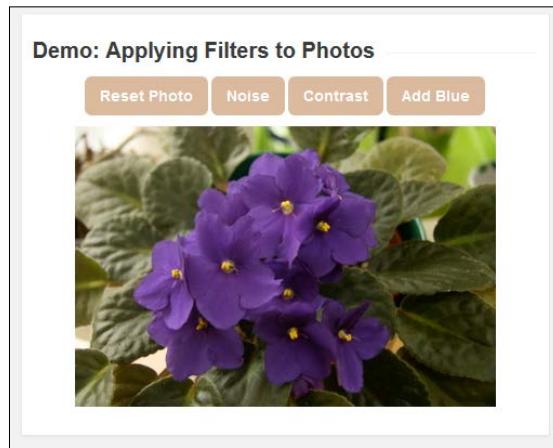


Remember – if you use Chrome, run this demo from within a local webserver, as suggested in the *Getting ready* section.



Let's begin:

1. Start by extracting the following files from the code download that accompanies this book. For this demo, we'll need the following files: `caman.html`, `flowers.jpg`, `usecaman.js`, `jquery.min.js`, and `usecaman.css`. Store the JavaScript files in the `js` subfolder, the CSS file in the `css` subfolder, the image within the `img` subfolder, and the HTML markup within the root area of our project folder.
2. Run the `caman.html` demo file. If all is well, we should see the following image appear:



3. Let's explore the jQuery required to operate the demo. If we peek inside `usecaman.js`, we'll see the following code. This is used to get a handle on the `<canvas>` element in our markup, before drawing the `flowers.jpg` image on it.

```
var canvas = $('#canvas');
var ctx = canvas[0].getContext("2d");
var img = new Image();
img.src = "img/flowers.jpg";
ctx.drawImage(img, 0, 0);
```

4. Digging a little deeper, we should see the following method – this handles the reset of the `<canvas>` element back to its original state; notice how the `drawImage()` method is used, which is key to manipulating images with different filters:

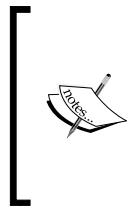
```
$reset.on('click', function(e) {
  e.preventDefault();
  var img = new Image();
```

```
img.src = "img/flowers.jpg";
ctx.save();
ctx.setTransform(1, 0, 0, 1, 0, 0);
ctx.clearRect(0, 0, canvas[0].width, canvas[0].height);
ctx.restore();
ctx.drawImage(img, 0, 0);
Caman('#maincanvas', 'img/flowers.jpg', function() {
    this.revert(false).render();
});
});
```

5. We then top it off with three different event handlers – these apply the relevant CamanJS filter:

```
$noise.on('click', function(e) {
    e.preventDefault();
    Caman('#maincanvas', 'img/flowers.jpg', function() {
        this.noise(10).render();
    });
});
```

Our simple demo only scratches the surface of what is possible with CamanJS. It is well worth having a look at the site in more detail, to get a feel for what can be achieved using the library. As a source of inspiration, take a look at the article by Carter Rabasa, which uses the library to create a Phonestagram application, based on the well-known Instagram site; it's available at <https://www.twilio.com/blog/2014/11/phonestagram-fun-with-photo-filters-using-node-hapi-and-camanjs.html>.



It's worth noting that CamanJS is able to handle HiDPI images with ease – all we need to do is set the `data-caman-hidpi` attribute in our code. Caman will automatically switch to using the hi-res version, if it detects that the device supports hi-res images. Note though, that rendering takes longer, due to the additional pixels being used.

Getting really creative

Cast your mind back to the beginning of this chapter, where I mentioned that CSS3 filters provide a convenient and lightweight means of manipulating images. Their use means that we can reduce the amount of work required when editing the images, and that should the images change in size or content, then it is much easier to update them.

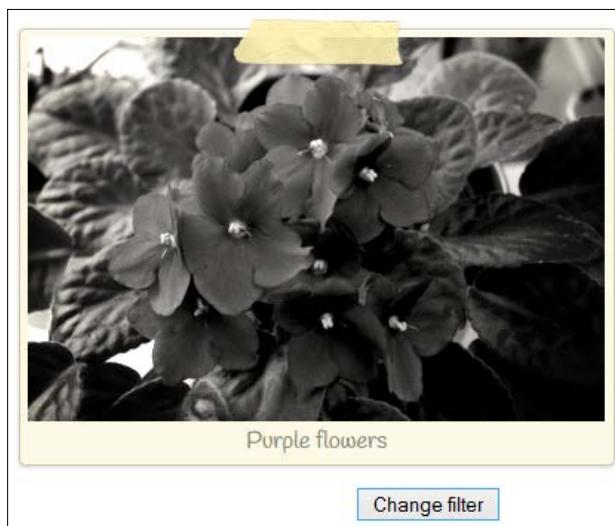
However, using CSS3 filters can only go so far – this is where jQuery takes over. To see why, let's work through another demo. This time, we'll use one of the more advanced preset filters that comes with CamanJS, and which would be difficult to achieve if we had to use CSS3 filters alone.

Remember – if you use Chrome, please run this demo from within a local web server, as suggested in the *Getting ready* section. Let's start:

1. For this demo, we need some files from the code download that accompanies this book. They are: `caman-advanced.css`, `caman-advanced.html`, `caman.full.js`, `jquery.min.js`, and `flowers.jpg`. Place each file in the relevant subfolders, and the HTML markup file in the root of our project area.
2. In a new file, add the following code to configure the CamanJS object to use the pinhole filter supplied with the library; save this as `caman-advanced.js` within the `js` subfolder:

```
$(document).ready(function() {  
    $("input").on("click", function() {  
        Caman("#caman-image", function () {  
            this.pinhole().render();  
        });  
    })  
});
```

3. If we preview the demo, we can see that the image now shows a pinhole camera effect when the **Change filter** button is clicked. Refer to the following image:



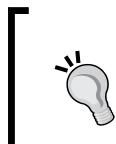
There are plenty of examples of more unusual filters on the CamanJS site. Head over to <http://camanjs.com/examples/> to view what is possible using the library.

Although we've concentrated on using CamanJS as our example (partially due to the breadth of what is possible to do with the library), there are other libraries available that offer similar filter functionality, but not all to the same level as CamanJS. Here are some examples to explore, to get you started:

- **VintageJS**: <https://github.com/rendro/vintageJS>
- **Hoverizr**: <https://github.com/iliasioviz/Hoverizr>
- **PaintbrushJS**: <http://mezzoblue.github.com/PaintbrushJS>
- **Colorimazer**: <http://colorimazer.tacyniak.fr/>

For those of you who prefer not to use open source, one example that you may like to explore is the JSManipulation library, which is available for sale from the CodeCanyon site at <http://codecanyon.net/item/jsmanipulate-jquery-image-manipulation-plugin/428234>.

Right, let's move on and really get stuck into something. So far, we've used plugins which will serve most purposes for us. But in some instances, we may find that we need to create our own filters manually, as existing filters are not available for our needs. Let's take a look at a couple in action, to see what is involved.



To see what is possible when using Caman, take a look at this article by Martin Angelov at <http://tutorialzine.com/2013/02/instagram-filter-app/>. It takes us through building an Instagram filter application, using jQuery, CamanJS, and the jQuery Mousewheel.

Creating simple filters manually

The key to creating our own filters (and as is the case with many prebuilt plugins), is to use the `<canvas>` element and familiarize ourselves with the `getImageData` method. We can use the latter to manipulate the color channels within each image to produce the desired effect.

We could spend time talking about using this method in detail, but I think it would be far better to see it in action. So let's dive in and use it to create a couple of filters manually, beginning with grayscaling an image.

Grayscaling an image

For the first demo of three, we're going to desaturate the colors in a copy of the `flowers.jpg` image that we've been using throughout this chapter. This will give it a grayscale appearance.



You may get cross-domain errors if running this demo locally. I would recommend running it within a local web server, as suggested in the *Getting ready* section.

Let's look at what we have to do:

1. Let's start by extracting a copy of `flowers.jpg`, `jquery.min.js`, `manual-grayscale.html`, and `manual-grayscale.css` from the code download that accompanies this book. Store the image in the `img` subfolder, the JavaScript file in the `js` subfolder, and the style sheet in the `css` subfolder; the HTML markup needs to be stored at the root of our project folder.
2. In a new file, go ahead and add the following code, saving it as `manual-grayscale.js` – this looks for each image set with a classname of `picture`, before calling the `grayscale` function to perform the magic:

```
$(window).load(function() {
    $('.picture').each(function() {
        this.src = grayscale(this.src);
    });
});
```

3. Add the following function immediately below the `$(window).load` method – this rewrites the image with a grayscale equivalent:

```
function grayscale(src) {
    var i, avg;
    var canvas = document.createElement('canvas');
    var ctx = canvas.getContext('2d');
    var imgObj = new Image();
    imgObj.src = src;
    canvas.width = imgObj.width;
    canvas.height = imgObj.height;
    ctx.drawImage(imgObj, 0, 0);
    var imgPixels = ctx.getImageData(0, 0, canvas.width,
        canvas.height);
    for(var y = 0; y < imgPixels.height; y++) {
        for(var x = 0; x < imgPixels.width; x++) {
            i = (y * 4) * imgPixels.width + x * 4;
```

```
    avg = (imgPixels.data[i] + imgPixels.data[i + 1] +
    imgPixels.data[i + 2]) / 3;
    imgPixels.data[i] = avg;
    imgPixels.data[i + 1] = avg;
    imgPixels.data[i + 2] = avg;
}
}
ctx.putImageData(imgPixels, 0, 0, 0, 0, imgPixels.width,
imgPixels.height);
return canvas.toDataURL();
}
```

4. If we run the demo at this point, we should see a copy of the image with the Polaroid effect border as before, but this time, it has been converted to a grayscale equivalent image, followed by the screenshot itself:



Before we continue with our next demo, there are a few key points to note, relating to the code we've just used. So let's spare a moment to cover these in more detail:

- Most of the work we've done uses the <canvas> element – this allows us to manipulate the image at a much finer detail than if we were using a plain JPG or PNG format image.
- In this instance, we've created the canvas element using plain JavaScript with the statement `document.createElement('canvas')`. Some may argue that mixing vanilla JavaScript with jQuery is bad practice. In this instance, I personally feel it provides a cleaner solution, as a context is not added automatically to <canvas> elements that are created dynamically with jQuery.

- `getImageData()` as a method is key to manipulating any image using this route. We can then work with each of the color channels, namely red, green, and blue, to produce the desired effect.

We can use this process to produce any number of different filters – how about a sepia one, for example? Let's take a look at how we can manually create such a filter. In this instance, we'll go one further and turn it into a mini plugin for reuse at a later date.

Adding a sepia tone

We've seen how straightforward it is to produce a color filter from the ground up – what about creating different types of filters? We can use similar techniques for other filters, so let's go ahead and create a sepia-based one, to complement the CSS3 version we used earlier in this chapter.



Remember – if you use Chrome, please run this demo from within a local web server, as suggested in the *Getting ready* section.

Let's make a start:

1. We'll start, as always, by extracting the relevant files from the code download that accompanies this book. For this one, we'll need the following: `jquery.min.js`, `flowers.jpg`, `manual-sepia.css`, and `manual-sepia.html`. Store them in the relevant subfolders of our project folder.
2. In a new file, we need to create our sepia plugin, so go ahead and add the following code, beginning with setting up the call to find all images with a classname of `.sepia`:

```
jQuery.fn.sepia = function () {
  $(window).load(function () {
    $('.sepia').each(function () {
      var curImg = $(this).wrap('<span />');
      curImg.attr("src", grayImage(this));
    });
  });
}
```

3. Next comes the all-important function – the `grayImage` function takes the image, draws it to a canvas, then manipulates each of the color channels in the image, before rendering it back to screen:

```
function grayImage(image) {
  var canvas = document.createElement("canvas");
  var ctx = canvas.getContext("2d");
```

```
canvas.width = image.width;
canvas.height = image.height;
ctx.drawImage(image, 0, 0);
var imgData = ctx.getImageData(0, 0, canvas.width,
    canvas.height);

for (var y = 0; y < imgData.height; y++) {
    for (var x = 0; x < imgData.width; x++) {
        var pos = (y * 4) * imgData.width + (x * 4);
        var mono = imgData.data[pos] * 0.32 + imgData.data[pos
            + 1] * 0.5 + imgData.data[pos + 2] * 0.18;
        imgData.data[pos] = mono + 50;
        imgData.data[pos + 1] = mono;
        imgData.data[pos + 2] = mono - 50;
    }
}
ctx.putImageData(imgData, 0, 0, 0, 0, imgData.width,
imgData.height);
return canvas.toDataURL();
}
};

$.fn.sepia();
```

4. Let's preview the results in a browser. If all is well, we should see our image with a nice sepia tone, as seen in the following image:



This version of the filter may look slightly different in terms of the code we've used, but most of this is due to reconfiguring it as a plugin, along with some changes in variable names. If we look carefully though, we would see that the same principles have been used in both examples, but have produced two different versions of the same image.



If you would like to learn more about using the `getImageData()` method, then take a look at the W3School's tutorial, which is available at http://www.w3schools.com/tags/canvas_getimagedata.asp.

Blending images

For our third and final demo, and to prove how versatile `getImageData()` can be, we're going to add a tint to the same flowers image that we've used throughout this chapter.

This demo is relatively straightforward to implement. We already have the framework in place, in the form of a plugin; all we need to do is swap out the nested `for...` block, and replace it with our new version. Let's start:

1. In a copy of `manual-sepia.js`, look for the following line at or around line **17**:

```
for (var y = 0; y < imgData.height; y++) {
```

2. Highlight and remove all the way down to line **25**. Replace it with the following code:

```
var r_weight = 0.44;
var g_weight = 0.5;
var b_weight = 0.16;
var r_intensity = 255;
var g_intensity = 1;
var b_intensity = 1;

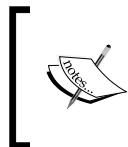
var data = imgData.data;
for(var i = 0; i < data.length; i += 4) {
    var brightness = r_weight * data[i] + g_weight * data[i +
        1] + b_weight * data[i + 2];
    data[i] = r_intensity * brightness; // red
    data[i + 1] = g_intensity * brightness; // green
    data[i + 2] = b_intensity * brightness; // blue
}
ctx.putImageData(imgData, 0, 0);
```

3. For now, save the file as `manual-sepia.js`, then preview `manual-sepia.html` in a browser. If all is well, we should see the image appear, but this time with a red tint, as in the following image:



The math used in this demo looks straightforward, but nonetheless may need a little explaining. It is a two-stage process, where we use the `_weight` variables to first work out the brightness levels, followed by using the `_intensity` variables to work out the relevant intensity level, before reapplying this to the appropriate color channel.

Getting to grips with the math required to build filters using this method can take time (and would be outside the scope of this book), but once you've understood the math, it opens up some real possibilities!



For convenience, I've reused the same files in this demo to prove that we can apply a specific color tint. In practice, we would need to rename the plugin name to better reflect the color being used (and that in this instance, wouldn't be sepia!).

We can of course take things even further. To do so can require some hardcore math, so won't be for the faint-hearted! If you fancy the challenge, then a good starting point is to learn about using **convolution masks**, which will look something like the following (this one being for blurring images):

```
[.1, .1, .1],  
[.1, .2, .1],  
[.1, .1, .1],
```

This will allow us to make some really complex filters, such as a Sobel filter (http://en.wikipedia.org/wiki/Sobel_operator), or even a Laplace filter (http://en.wikipedia.org/wiki/Discrete_Laplace_operator#Implementation_in_Image_Processing) – be warned: the math is really hardcore! To bring it back down to something a little easier, have a look on Google. Following are some useful starting points:

- <http://halfpapstudios.com/blog/2013/01/canvas-convolutions/>
- <https://thiscouldbebetter.wordpress.com/2013/08/14/filtering-images-with-convolution-masks-in-javascript/>
- <http://beej.us/blog/data/convolution-image-processing/convolution.js>

Let's change track! We've applied a number of filters using different means to our image, but has anyone noticed how abrupt the effect can be? A more pleasing route is to animate the transition process. Let's take a look at how we can achieve this, using the **cssAnimate** library.

Animating images with filters

Okay, so we've covered a number of different ways of applying filters to manipulate the appearance of images. Before we move on and take a look at some practical examples, let's pause for a moment.

Did anyone notice how when using jQuery, we lose the ability to gradually transition from one state to another? Transitioning is just one way of providing a nice touch to any change of state – after all, it is far easier on the eye to gradually change state, than to see an abrupt switch!

We could spend time crafting a solution from the ground up using jQuery. However, a more prudent solution would be to use a plugin for this purpose.

Introducing cssAnimate

Enter cssAnimate! This little gem by Clemens Damke generates the necessary CSS3 styles to animate a state change, but falls back to using jQuery's `animate()` method handler, if support is not available. The plugin is available for download from <http://cortys.de/cssAnimate/>. Although the site indicates a minimum requirement of jQuery 1.4.3 or above, it works with no noticeable issues when used with jQuery 2.1.

Let's take a look at a screenshot of what we're going to produce:



Let's start:

1. We'll begin with extracting the following files from the code download that accompanies this book: `cssanimate.html`, `cssanimate.css`, `flowers.jpg`, `jquery.min.js`, and `jquery.cssanimate.min.js`.
2. Save the JavaScript files to the `js` subfolder, the image to the `img` folder, the CSS file to the `css` subfolder, and the HTML markup to the root folder of our project area.
3. In a separate file, add the following code, which animates a change of hue-rotate filter:

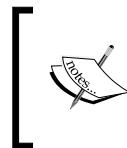
```
$ (document) . ready (function () {  
    $ ("input [name='css']") . on ("click", function () {  
        $ ("img") . cssAnimate ({filter: hue-rotate(50deg), -webkit-filter: hue-rotate(50deg)}, 500, "cubic-bezier(1,.55,0,.74)") ;  
    })  
});
```

4. If all is well, we should see the flowers appear to turn a shade of dark pink when clicking on the **Change filter using CSS** button, as shown at the beginning of our exercise.

At first appearance, the only change we will see is the transition to a darker shade of pink in our image. However, the real change will show if we inspect our code using a DOM Inspector, such as Firebug:

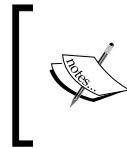
```
element.style {
    filter: hue-rotate(50deg);
    transition-duration: 0s;
    transition-property: filter, -webkit-filter, -webkit-filter, -moz-filter, -o-filter, -ms-filter,
    -khtml-filter;
    transition-timing-function: cubic-bezier(1, 0.55, 0, 0.74), cubic-bezier(1, 0.55, 0, 0.74),
    cubic-bezier(1, 0.55, 0, 0.74), cubic-bezier(1, 0.55, 0, 0.74), cubic-bezier(1, 0.55, 0, 0.74),
    cubic-bezier(1, 0.55, 0, 0.74), cubic-bezier(1, 0.55, 0, 0.74);
```

The beauty about this library is that despite it being a few years old, it still appears to work well with modern versions of jQuery. It opens up some real avenues that we can explore, in terms of the transition animations that we can use.



Transition support is almost 100 percent across the main browsers, save for Opera Mini. To get an up-to-date picture, it's worth checking the Can I Use site at <http://caniuse.com/#feat=css-transitions>.

Although the number of built-in animations is limited within cssAnimate, it does at least include support for cubic-bezier values. Matthew Lein has produced a file that contains a number of cubic-bezier equivalents for well-known easing effects; this is available from <https://github.com/matthewlein/Ceaser/blob/master/developer/ceaser-easings.js>. We can use this to provide the values that can be dropped into our animation to produce the desired effect. Alternatively, we can design our own cubic-bezier easing effect using a site such as <http://cubic-bezier.com> – this provides similar values that can be used in our animation in the same way.



As an aside – I came across this neat little demo when researching for this book: <http://codepen.io/dudleystorey/pen/pKoqa>. I wonder if we could use cssAnimate to produce a similar effect?

Okay – enough of filters for the moment! Let's change focus and dive into something a little more practical. How many of you have had to sign for something online, using an electronic signature? It's a great effect to incorporate, should the circumstances require it. We're going to take a look at how, but extend it further, so that we can save the image for later use.

Creating a signature pad and exporting the image

Now that we've seen how we can manipulate images, let's turn our attention to something more fundamental; capturing images drawn on canvas elements.

As we move more and more into a digital world, there will be occasions when we are asked to "sign" a document electronically, using our computer. It does mean that we shouldn't consider signing anything the morning after a heavy night out, but worse things can happen...! That in mind, let's take a look at how we can capture the image, once the document has been signed.

For this demo, we're going to use the Signature Pad plugin for jQuery, by Thomas Bradley. The plugin is available from <http://thomasjbradley.ca/lab/signature-pad>. We're going to take it a step further – instead of just signing our name, we will provide an option to save the output as a PNG file, using the `canvas.toDataURL()` method.



Remember – if you use Chrome, please run this demo from within a local web server, as suggested in the *Getting ready* section.



Let's start:

1. We'll begin by downloading the CSS and HTML markup files that are required for this demo, from the code download that accompanies this book. Go ahead and extract the signature pad folder and save it to the project area.
2. Next, add the following code to a new file – save it as `signaturepad.js`, within the `js` subfolder of our demo folder:

```
$(document).ready(function() {  
    $('.sigPad').signaturePad();  
    var canvas = $('#canvas')[0], ctx = canvas.getContext('2d');  
  
    $('#download').on('click', function() {  
        saveImage();  
        downloadCanvas(this, 'canvas', 'signature.png');  
    });  
  
    function saveImage() {  
        var api = $('.sigPad').signaturePad();  
        var apitext = api.getSignatureImage();  
        var imageObj = new Image();
```

```
imageObj.src = apitext;
imageObj.onload = function() {
    ctx.drawImage(imageObj, 0, 0);
};

function downloadCanvas(link, canvasId, filename) {
    link.href = $(canvasId)[0].toDataURL();
    link.download = filename;
}
});
```

[ There is a version of this file already in the code download; extract and rename `signaturepad-completed.js` to `signaturepad.js`, then store in the same js folder as outlined in this demo.]

3. If we preview the results in a browser, we should see a signature pad displayed, as shown in the following screenshot:



In this screenshot, I've already added my name. Try clicking on **Draw It** and then drawing your name – beware, it takes a steady hand! Next, click on the link. If all is well, we will be prompted to open or save file named `signature.png`. Opening it up in a suitable graphics package confirms that the signature was saved correctly. Refer to the following image:



Although this is a relatively simple demo, it opens up some real possibilities. Outside of the signature plugin we've used, the key to this demo is two-fold: the use of a `<canvas>` element to capture the drawn signature, and the `.toDataURL()` method used to convert the contents of the canvas element to a data URI, which contains a representation of the image in PNG format (by default).

We first get a handle of, then draw out the image onto, a canvas element. As soon as the download event handler is fired, it converts the image to a data URI representation, then renders it into a format that we can save for later use.



If you would like to learn more about the `toDataURL()` method, then Mozilla's Developer Labs have a good article, which is available at <https://developer.mozilla.org/en-US/docs/Web/API/HTMLCanvasElement.toDataURL>.

Let's put this technique to good use and combine it with using the webcam and the image manipulation techniques we covered at the start of this chapter. This allows us to get really crazy; fancy having some fun with capturing and changing webcam images?

Capturing and manipulating webcam images

In our second and final demo for this chapter, we're going to have some fun with a webcam – one of the ways we can acquire and manipulate images is to source them from a laptop or stand-alone webcam.

The key to this demo lies in the use of `getUserMedia`, which allows us to control audio or video feeds. This is a relatively young API, which requires use of vendor prefixes to ensure full support. As with other APIs, the need for them will disappear over time, so it is worth checking <http://caniuse.com/#search=getusermedia> regularly to see if support has been updated and the need for prefixes removed.

This demo will bring together some of the concepts we've explored, such as applying filters, saving canvas images to file, and controlling a webcam. To operate this demo correctly, we will need to run it from an HTTP protocol address and not `file://`. For this, you will either need some web space available, or to use a local webserver such as WAMP (for PC - <http://www.wampserver.com/en>), or MAMP (for Mac, and now PC, from <http://www.mamp.info/en/>).

Okay, assuming this is in place, let's start:

1. We'll start by extracting the `webcam` demo folder from the code download that accompanies this book. It contains the styling, markup, and a copy of the jQuery library that is needed for this demo.
2. Once extracted, upload the whole folder to your web space. I will assume you are using WAMPServer, so this will be the `/www` folder; if you are using something different, then please alter accordingly.
3. We need to add the jQuery magic that is needed to make this demo work. In a new file, go ahead and add the following code; we'll work through it in sections, beginning with assigning variables and a filter array:

```
$(document).ready(function() {  
    var idx = 0;  
    var filters = ['grayscale', 'sepia', 'blur', 'saturate', ''];  
  
    var canvas = $("canvas")[0], context =  
        canvas.getContext("2d"),  
    video = $("video")[0], localStream, videoObj = { "video":  
        true }, errBack = function(error) {  
        console.log("Video capture error: ", error.code);  
    };
```

4. The first function handles the paging through of the filters. We cycle through the filter names stored within the filter array. If there is a corresponding style rule within the style sheet, then the following is applied to the canvas image:

```
function changeFilter(e) {  
    var el = e.target;  
    el.className = '';  
    var effect = filters[idx++ % filters.length];  
    if (effect) {
```

```
    el.classList.add(effect);
}
}
```

5. Next, we need to get an instance of `getUserMedia`, which we use to control the webcam. As this is still a relatively young API, we are obliged to use the vendor prefixes:

```
navigator.getUserMedia = (navigator.getUserMedia ||
  navigator.webkit GetUserMedia || navigator.mozGetUserMedia ||
  navigator.msGetUserMedia);
```

6. The first of several event handlers, the `#startplay` button is the most important. Here we capture the webcam source, then assign it to the video object and generate the URL that references our content. Once assigned, we start the video feed playing, which allows us to view the content on screen:

```
$("#startplay").on("click", function(e) {
  if (navigator.getUserMedia) {
    navigator.getUserMedia(videoObj, function(stream) {
      video.src = window.URL.createObjectURL(stream);
      localStream = stream;
      video.play();
    }, errBack);
  }
});
```

7. We then need to assign some event handlers. In order, the following handle requests to take a snapshot of the image, stop the video, change the filter, and download a copy of the snapshot image:

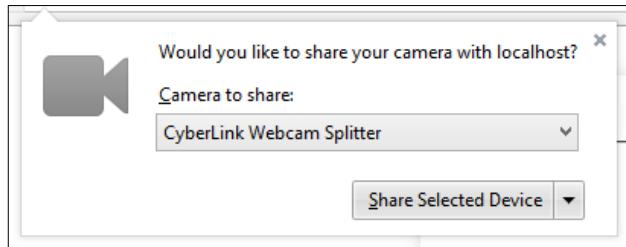
```
$("#snap").on("click", function() {
  context.drawImage(video, 0, 0, 320, 240);
});

$("#stopplay").on("click", function(e, stream) {
  localStream.stop();
});

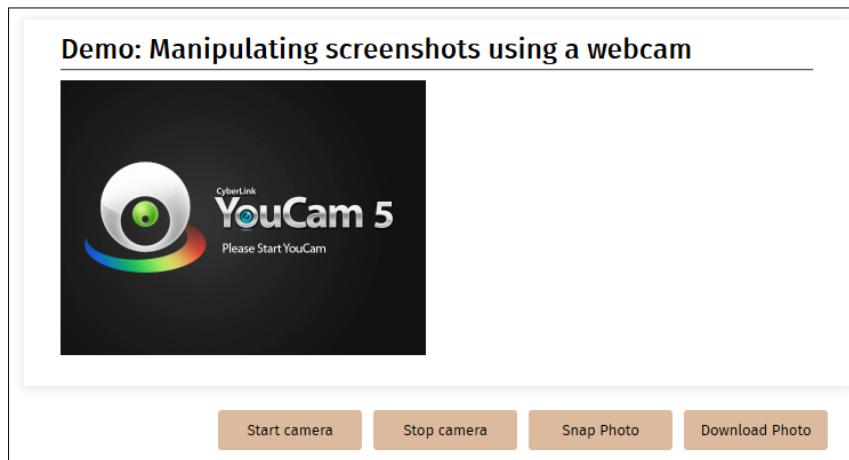
$('#canvas').on('click', changeFilter);

$("#download").on('click', function (e) {
  var dataURL = canvas.toDataURL('image/png');
  $("#download").prop("href", dataURL);
});
});
```

8. Save the file as `webcam.js` within the `js` subfolder of the `webcam demo` folder that we uploaded earlier in this demo.
9. At this point, we can try running the demo within a browser. If all is well, we will first get a request to allow the browser access to the webcam (for security reasons), as shown in the following image:



10. This is then followed by an initialization of the camera. It starts with a placeholder image, as shown next; this will then display the live feed within a few moments:



As this point, we can have all sorts of fun. Try clicking on **Snap Photo** to take a snapshot of yourself; this will appear to the right of the live feed. If we click on this image, it will cycle through several filters that we've set up in the style sheet, and reference using the following line in `webcam.js`:

```
var filters = ['grayscale', 'sepia', 'blur', 'saturate', ''];
```

Hang on – did anyone notice something about the image we get when clicking on the **Download Photo** button? The keen-eyed amongst you will soon spot that it is a copy of the original image, before filters have been applied.

The reason for this is that the filters are set within CSS – naturally, they only have any effect when displayed in the browser window! To fix this, we need to alter our download event handler. We can use the CamanJS library that we explored earlier to apply some basic filters, such as the Sunrise effect that comes with the library.

To do this, alter the `#download` event handler to show the following code:

```
$("#download").on('click', function (e) {  
    e.preventDefault();  
    Caman('#canvas', function(){  
        this.sunrise();  
        this.render(function() { this.save("webcam-photo.png"); });  
    });  
});
```

Now try saving a copy of the screenshot. While it doesn't force a download to your desktop, it will nonetheless display an image in the browser now that shows the Sunrise filter applied to it.

We've only scratched the surface of what is possible when using `getUserMedia` – it is well worth exploring this online to learn more. A good starting point is the article on the Mozilla Developer Network, which is available at <https://developer.mozilla.org/en-US/docs/NavigatorUserMedia.getUserMedia>. Note – `getUserMedia` is not supported in IE11 or below, so you will need to use a polyfill library such as `getUserMedia.js` by Addy Osmani, which is available for download at <https://github.com/addyosmani/getUserMedia.js>.

[As an aside, I had considered including something on using the reveal.js library to control a simple image gallery using hand gestures in this book, as shown at <http://www.chromeexperiments.com/detail/gesture-based-revealjs/>. The unfortunate thing is that the code isn't rock solid, and hasn't been updated for some time. I'd be intrigued to hear what your thoughts are. It's a great way of showing off a slick means of presenting content, but needs more work!]

Finishing up

Before we round up this chapter, it's worth pausing for a moment to consider the implications of some of the techniques that we've covered in this chapter.

The purists may question the need to use jQuery to apply filters, particularly if all we need to do is to use a method such as `.addClass()` or even `.toggleClass()` to apply or remove a specific filter. The flip side of this is that this book is of course about jQuery, and that this is what we should concentrate on using, even at the cost of the apparent delay in showing some of the filter effects we've used.

The short answer to this will depend on you – anyone can write jQuery code to a greater or lesser extent, but the difference between an average and a good developer is not just in writing code.

The real difference lies partially in making the right choices. jQuery is frequently seen as the easy option, particularly as it provides the widest range of support. We can create any kind of filter to fit our needs, but it is always at the expense of processing power – we cannot get away from the fact that manipulating the canvas element takes a lot of resources, so is slow to complete. This is no better if high definition images are used (as we noted back in the *Applying filters with CamanJS* section) – indeed, it's even slower, given that more pixels need to be processed!

The upshot of this is that we need to carefully consider what filters we need to apply, and whether we can simply use CSS3 filters to fulfill our needs. It is true that these may not provide a solution for all our needs, but support is changing. We should really consider using jQuery filters where the delay isn't an issue, and the application won't be used on a mobile platform (due to the resources required to process each pixel!).

Summary

Manipulating images is one of the paradoxes within jQuery – we can use CSS3 filters to produce concise effects with little effort, but be limited to what CSS3 filters can offer; or we can produce any filter we desire, but at the expense of the processing resources required to manipulate images at a pixel level! In this chapter, we've covered a lot of information, so let's take a moment to recap what we've learnt.

We kicked off with a look at adding filters using CSS3, and saw how easy it is to apply these to an image. We then moved onto examining a different technique of blending images using CSS3, before turning our attention to examining jQuery image plugins.

Manipulating Images

We spent a little time exploring some of the basic options to apply filters, and then created our own jQuery based filters. We then switched to looking at how we can animate the transition into using a filter, to help provide a smoother crossover, before finishing with a look at creating basic demos using a signature pad and a webcam, as a means of capturing images using jQuery.

We then rounded up the chapter with some final thoughts on when we should be using CSS3 filters or jQuery, as a means of emphasizing that anyone can write code, but that good developers know which tool to use at the right time in their development process.

In the next chapter, we're going to expand on the use of plugins, with a look at taking our plugin development skills to the next level.

11

Authoring Advanced Plugins

Throughout this book, a common theme has been to use plugins – it's now time to create one!

There are literally thousands of plugins available for use, from ones that might only be a few lines long, to those running into several hundred lines. I'm a great believer in the phrase "where there's a will, there's a way" – it could be argued that plugins satisfy that will, and provide a way of resolving a need or a problem for a user.

Over the next few pages, we'll take a look at developing an advanced plugin from start to finish. Rather than concentrating only on the construction (as such), we'll take a look at some of the tips and tricks we can use to help push our development skills further when working with plugins. We'll cover best practices, and look at some areas where you can improve your current coding skills. Throughout the next few pages, we will cover a number of topics, which will include:

- Best practices and principles
- Detecting signs of a poorly developed plugin
- Creating design patterns for jQuery plugins
- Designing an advanced plugin and making it available for use

Ready to start?

Detecting signs of a poorly developed plugin

Imagine the scenario, if you will – you spend weeks developing a complex plugin, which does everything but the kitchen sink, and leaves anyone watching in awe.

Sounds like the perfect nirvana, right? You publish it on GitHub, create an awesome website, and wait for users to roll in and download your latest creation. You wait... and wait...but get a grand total of zero customers. Okay...so what gives?

Anyone can write code, as I always say. The key to becoming a better jQuery plugin developer is understanding what makes a good plugin, and knowing how to put that into practice. To help with this, let's take a moment to look at some pointers we can use to spot when a plugin is likely to fail:

- You're not making a plugin! The accepted practice is to use one of a handful of plugin patterns. If you're not using one of these patterns (such as the one shown next), then there is a good chance that take-up of your plugin is likely to be low.

```
(function($, window, undefined) {
    $.fn.myPlugin = function(opts) {
        var defaults = {
            // setting your default values for options
        }
        // extend the options from defaults with user's options
        var options = $.extend(defaults, opts || {});
        return this.each(function(){ // jquery chainability
            // do plugin stuff
        });
    })(jQuery, window);
```

- Although we've defined the `undefined` in the parameters, we are only using `$` and `window` in the self-invoking function. It shields the plugin from being passed malicious values to `undefined`, as it will remain as `undefined` within the plugin.
- You spend time writing code, but miss one of the key elements – preparing documentation! Time and again, I see plugins that have minimal or non-existent documentation. It makes it hard to understand the plugin's makeup, and work out how to use it to its full potential. There are no hard-and-fast rules with documenting, but it is generally accepted that the more the better, and that this should be both inline and external (in the form of a `readme` or `wiki`).
- Continuing with the theme of a lack of suitable documentation, developers will be turned off by plugins that have hardcoded styling, or which are too inflexible. It's up to us to consider all possible needs, but to determine if we're going to provide a solution for a particular need. Any styling that is applied to the plugin should either be made available via plugin options, or as a specific class or selector ID within the style sheet – putting it in line is considered bad practice.

- If your plugin requires too much configuration, then this is likely to turn people off. While a larger, more complex plugin should clearly have more options available to end users, there is a limit to what should be provided. Conversely, every plugin should at least have a no-argument default behavior set; users will not appreciate having to set multiple values just to get a plugin working!
- A big turn-off for end users is plugins that don't provide some form of example. At an absolute minimum, a basic "hello world" type example, should always be provided, with a minimal configuration defined. Those plugins that provide more involved examples, or even examples that work with other plugins, are likely to attract more people.
- Some plugins fail for basic reasons. These include: not providing a changelog or using version control, not working across multiple browsers, using an outdated version of jQuery or including it when it isn't really needed (dependencies are too low), or not providing a minified version of the plugin. With Grunt, there is no excuse! We can automate a large part of the basic admin tasks that are expected of developers, such as testing, minifying the plugin, or maintaining version control.
- Finally, plugins can fail for one of two simple reasons: either they are too clever and try to achieve too much (making them difficult to debug), or too simple, where the dependencies on jQuery as a library are not enough to warrant including it.

Clearly a lot to think about! While we can't predict if a plugin will be successful or if take up will be low, we can at least try to minimize the risk of failure by incorporating some (or all) of these tips into our code and development workflow.

At a more practical level though, we can opt to follow any one of a number of design patterns, to help give structure and consistency to our plugin. We touched on this back in *Chapter 3, Organizing Your Code*. The beauty is that we are free to use similar principles with jQuery plugins too! Let's take a moment to consider some possible examples, before using one to develop a simple plugin.

Introducing design patterns

If you've spent any time developing code in jQuery, then it is very likely that you've created one or more plugins; these can technically range from just a handful of lines to something more substantial.

Over time, there is a risk that amending code in plugins can lead to content becoming unwieldy and difficult to debug. One way of dealing with this is to use design patterns. We covered this back in *Chapter 3, Organizing Your Code*. Many of the same principles can equally apply to plugins, although the patterns themselves will of course be different. Let's consider a few examples.

The most basic pattern is **A Lightweight Start**, which will suit those who have developed plugins before, but are new to the concept of following a specific pattern. This particular pattern is based around common best practices, such as using a semicolon before invoking the function; it will pass in standard arguments such as `window`, `document`, and `undefined`. It contains a basic default object which we can extend, and adds a wrapper around the constructor to prevent issues with multiple installations.

At the opposite end, we can always try working with the **Complete Widget Factory**. Although it is used as the basis for jQuery UI, it can also be used to create standard jQuery plugins. This pattern is perfect for creating complex, state-based plugins. It contains comments for all the methods used, to help ensure that logic fits into your plugin.

We've also covered the concept of namespacing, or adding a specific name to avoid collisions with other objects or variables within the global namespace. Although we might use namespacing within our code, we can equally apply it to plugins too. The great thing about this particular pattern is how we can check for its existing instances; if the name doesn't exist then we are free to add it, otherwise we can extend an existing plugin with the same namespace.

These are three of the plugin patterns that are available for use; a question I am sure will arise, though, is which one to use? As with many things, there is no right or wrong answer; it will depend on circumstances.



A list of the most common plugin design patterns is available at
<https://github.com/jquery-boilerplate/jquery-patterns>.

Creating or using patterns

If you're new to using plugin design patterns, then A Lightweight Start is the best place to begin. There are three key aspects to using any plugin pattern, or designing your own:

- **Architecture:** This defines the rules of how your components should interact.
- **Maintainability:** Any written code should be easily extendable and improvable. It should not be locked down from the start.

- **Reusability:** How often can you reuse your existing code? The more it can be reused, the more time it will save, and it will also be easier to maintain.

The important thing about using patterns is that there isn't a single right answer. It all boils down to which pattern most closely fits your needs. The best way to gauge which pattern fits best is to try them. Over time, experience will give you a clear indication as to which pattern works best for a given scenario.



For a good discussion on the pros and cons of using a particular plugin pattern, head over to the article by Smashing Magazine at <http://www.smashingmagazine.com/2011/10/11/essential-jquery-plugin-patterns/>. It may be a few years old, but many of the points still hold value.

Anyway, let's get back to the present! There is no time better than now to start gaining experience, so let's take a look at the jQuery Lightweight Boilerplate pattern. This implements the Singleton/Module design pattern. It helps developers to write encapsulate code that can be kept away from polluting the global namespace.

Over the next few pages, we'll be developing a tooltip plugin. We'll start with a typical build that doesn't use any pattern, before modifying it to use the Lightweight Boilerplate style. We'll then delve into a few tips and tricks that will help us consider the bigger picture, and hopefully make us better developers.

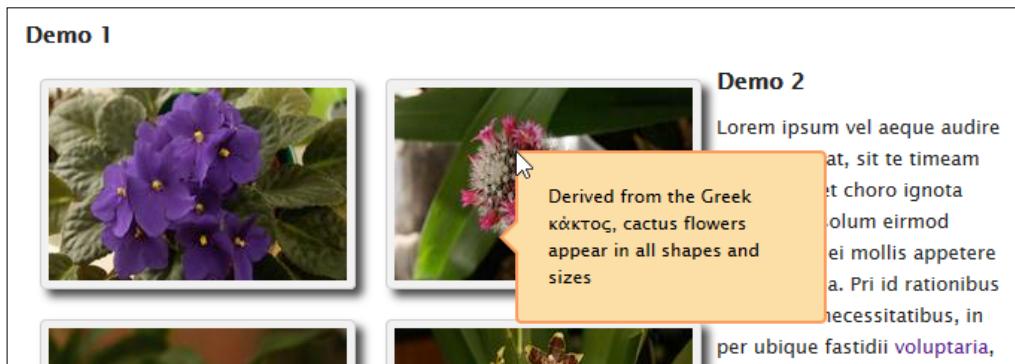
Designing an advanced plugin

Right – enough chitchat! Let's get down and dirty with some code! Over the next few pages, we're going to spend some time developing a plugin that displays some simple tooltips on a page.

Okay, before you all groan and say "not another tooltip plugin...!", there is a good reason for choosing this functionality. All will become clear once we've developed the first version of our plugin. Let's make a start - we'll begin with a brief look at creating our plugin:

1. For this demo, we'll need the entire code folder for this chapter from the code download that accompanies this book. Go ahead and extract it, saving it to our project area.

2. In the folder, run the `tooltipv1.html` file, which contains a grid of six images, along with some dummy text. Hover over the images in turn. If all is well, it will show a tooltip:



At this point you're probably wondering how all the code hangs together. It's a valid question...but we're going to break with tradition, and not examine it. Instead, I want to concentrate on redesigning the code to use boilerplate formatting, which will help make it easier to read, debug, and extend in the future. Let's consider what this means for our plugin.

Rebuilding our plugin using boilerplate

Hands up if you've not heard of boilerplating? Chances are that you may or may not have come across such examples as Bootstrap (<http://www.getbootstrap.com>), or even HTML5 Boilerplate (<https://html5boilerplate.com/>). To help you get familiar with the term, it is based on a simple idea of using a template to help structure code. It doesn't mean that it will write it for us (shame – we could earn millions for doing nothing, chuckle!), but it helps to save time by reusing a framework to rapidly develop code, such as full websites or even jQuery plugins.

For our next demo, we're going to rework our plugin using the jQuery Boilerplate templates available from <https://github.com/jquery-boilerplate/jquery-patterns>. As is often the case with the Internet, some kind soul has already created a good example of a tooltip using this technique, so we'll adapt it for our needs.



If you are interested in learning more about the jQuery Boilerplate plugin pattern, you may like to look at *Instant jQuery Boilerplate for Plugins*, by Jonathan Fielding, available from Packt Publishing.

The plugin example we'll use is by Julien G, a French web developer. The original is available via JSFiddle at <http://jsfiddle.net/molokoloco/DzYdE/>.

1. Let's start (as always), by extracting a copy of the code folder for this chapter from the code download for this book. If you already have it from the previous exercise, then we can use that instead.
2. Navigate to the `version 2` folder, then preview `tooltipv2.html` in a browser. If all is well, we should see the same set of images as in the previous example, with the same styling applied for the tooltips.

At face value, it would seem that nothing has changed – this in itself is actually a good indicator of success! The real changes though are in `tooltipv2.js`, within the `js` subfolder under the `version 2` folder. Let's go through this step by step, beginning with declaring variables:

1. We start with declaring properties for the `jQuery`, `document`, `window`, and `undefined`. You might ask why we are passing in `undefined` – it's an excellent question: this property is mutable (meaning it can be changed). Although it was made non-writable in ECMAScript 5, not using it in our code means it can remain `undefined` and prevent malicious code attempts. Passing the remaining three properties makes it quicker to reference within our code:

```
(function($, window, document, undefined) {  
    var pluginName = 'tooltip', debug = false;
```

2. Next up, the internal methods. We're creating them as methods within the `internal` object; the first takes care of positioning the tooltip on screen, while `show` and `hide` controls the visibility of the tooltip:

```
var internal = {  
    reposition: function(event) {  
        var mousex = event.pageX, mousey = event.pageY;  
  
        $(this)  
            .data(pluginName) ['tooltip']  
            .css({top: mousey + 'px', left: mousex + 'px'});  
    },  
  
    show: function(event) {  
        if (debug) console.log(pluginName + '.show()');  
        var $this = $(this), data = $this.data(pluginName);  
  
        data['tooltip'].stop(true, true).fadeIn(600);  
        $this.on('mousemove.' + pluginName,  
            internal.reposition);
```

```
        } ,  
  
        hide: function(event) {  
            if (debug) console.log(pluginName + '.hide()');  
            var $this = $(this), data = $this.data(pluginName);  
            $this.off('mousemove.' + pluginName,  
                     internal.reposition);  
            data['tooltip'].stop(true, true).fadeOut(400);  
        }  
    };
```

3. We move on to the external methods. Up first from within the `external` object, comes the `init` function, to initialize our plugin and render it on screen. We then call the `internal.show` and `internal.hide` internal methods when moving over an element with an instance of the `.tooltip` class:

```
var external = {  
    init: function(options) {  
        if (debug) console.log(pluginName + '.init()');  
  
        options = $.extend(  
            true, {},  
            $.fn[pluginName].defaults,  
            typeof options == 'object' && options  
        );  
  
        return this.each(function() {  
            var $this = $(this), data = $this.data(pluginName);  
            if (data) return;  
  
            var title = $this.attr('title');  
            if (!title) return;  
            var $tooltip = $('                class: options.class,  
                text: title  
            }).appendTo('body').hide();  
  
            var data = {  
                tooltip: $tooltip,  
                options: options,  
                title: title  
            };  
  
            $this.data(pluginName, data)  
                .attr('title', '')
```

```
        .on('mouseenter.' + pluginName, internal.show)
        .on('mouseleave.' + pluginName, internal.hide);
    });
},
},
```

4. The second external method handles the updating of the tooltip text, using the `.data()` method:

```
update: function(content) {
  if (debug) console.log(pluginName +
    '.update(content)', content);
  return this.each(function() {
    var $this = $(this), data =
      $this.data(pluginName);
    if (!data) return;
    data['tooltip'].html(content);
  });
},
},
```

5. We round up the methods available in our plugin with a `destroy()` handler. This stops a selected tooltip from displaying, and removes the element from code:

```
destroy: function() {
  if (debug) console.log(pluginName + '.destroy()');

  return this.each(function() {
    var $this = $(this), data =
      $this.data(pluginName);
    if (!data) return;

    $this.attr('title', data['title']).off('.' +
      pluginName)
      .removeData(pluginName);
    data['tooltip'].remove();
  });
}
};
```

6. Last, but by no means least is our plugin initiator. This function simply maps method names to valid functions in our plugin, or degrades gracefully if they don't exist:

```
$.fn[pluginName] = function(method) {
  if (external[method]) return external[method]
  apply(this, Array.prototype.slice.call(arguments, 1));
  else if ($.type(method) === 'object' || !method)
```

```
        return external.init.apply(this, arguments);
    else $.error('Method ' + method + ' does not exist on
        jQuery.' + pluginName + '.js');
};

$.fn[pluginName].defaults = {
    class: pluginName + 'Element'
};
}) (window.jQuery);
```

The key takeaway though from this demo is not the specific functions that we can use, but the format used to produce our plugin.

Anyone can write code, but use of a boilerplate pattern such as the one we've used here will help improve readability, make it easier to debug, and increase opportunities when extending or upgrading functionality at a later date. Remember, if you write a plugin and don't revisit it for a period of time (say 6 months); then the acid test is how much you can work out from the well-structured code, without needing lots of documentation. If you can't do that, then you need to revisit your coding!

Let's move on. Remember when I mentioned there was a good reason for choosing to use a tooltip plugin as the basis for our examples? It's time to reveal why...

Converting animations to use CSS3 automatically

We've built a tooltip plugin which uses a touch of animation to fade in and out when hovering over elements marked with the `.tooltip` class. Nothing wrong in that - the code works perfectly well, and is an acceptable way of displaying content...right? Wrong! As you should know by now, we can definitely do better. Here's why I chose the tooltip as our example.

Remember how I mentioned back in *Chapter 6, Animating in jQuery*, that we should consider using CSS3 styling to control our animation? Well, here's a perfect example: we can easily change our code to force jQuery to use CSS3 where possible, or fall back to using the library for older browsers.

The trick behind it is in one line:

```
<script src="js/jquery.animate-enhanced.min.js"></script>
```

To see how easy it is, follow the next steps:

1. In a copy of `tooltipv2.html`, add this line as indicated:

```
<script src="js/jquery.min.js"></script>
<script src="js/jquery.animate-enhanced.min.js"></script>
<script src="http://code.jquery.com/ui/1.11.3/jquery-
ui.min.js"></script>
<script src="js/jquery.quicktipv2.js"></script>
<script src="js/tooltipv2.js"></script>
```

2. Preview the results in a browser. If all is well, we should see some slight changes in how the tooltip reacts. However, the real change is evident when viewing the code for the tooltip itself, from within a DOM Inspector such as Firebug:

```
<div class="arrow_box" style="display: none; top: 264px; left:
158px; transition-property: all; transition-duration: 0s;
transition-timing-function: cubic-bezier(0.25, 0.1, 0.25,
1);">Thailand Paradise Homes Property Website: Custom built
property website and database</div>
```

If we look in the computed styles half of Firebug, we can see the styles being assigned to the tooltip element:

```
element.style {
    display: none;
    left: 158px;
    top: 264px;
    transition-duration: 0s;
    transition-property: all;
    transition-timing-function: cubic-bezier(0.25, 0.1, 0.25, 1);
}
```

A simple change to make, but hopefully one we can see making a significant improvement in performance. In this instance, we're using a plugin to force jQuery to use CSS3 styling in place of standard jQuery based animations.

The key message here, though is that we should not, as developers, feel we are constricted to using jQuery to provide our animations. While it might be a necessary evil for managing complex motions, we should still consider using it for those less ornate instances.

Working with CSS-based animations

Hmm – it's at this point that a question comes to mind: surely, if we're using modern browsers, why do we need to rely on using jQuery-based animations at all?

The answer is simple – in short, it depends on the circumstances. The long answer though is that for modern browsers, we don't need to rely on using jQuery to provide our animations. Only if we're forced to provide support for old browser versions (such as IE6), do we need to use jQuery.

The likelihood though should be low. If we need to, then we should really be asking ourselves if we're making the right move, or whether support should be degraded gradually using something like Modernizr.

That all said – let's go through the following steps to understand what we need to do to use CSS3 in place of jQuery-based animations:

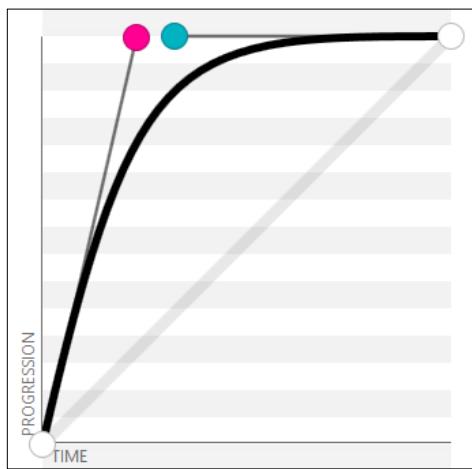
1. In a copy of `tooltipv2.css`, add the following CSS style at the bottom of the file – this will be our transition effect for the tooltip:
`div.arrow_box { transition-property: all; transition-duration: 2s; transition-timing-function: cubic-bezier(0.23, 1, 0.32, 1); }`
2. Open a copy of `jquery.quicktipv2.js`, then first comment out the line:
`data['tooltip'].stop(true, true).fadeIn(600);`
Add the following line in its place:
`data['tooltip'].css('display', 'block');`
3. Repeat the same process, but this time for the line:
`data['tooltip'].stop(true, true).fadeOut(400);`
Add this next line as its replacement:
`data['tooltip'].css('display', 'none');`
4. Save the files. If we preview the results of the change in a browser, we should see the tooltip appear to slide in and hover over the selected image.

The effect looks very smooth. While it doesn't fade in or out, it still provides an interesting twist to how tooltips would normally appear in a page. It does raise an interesting question – what effect should we use? Let's take a timeout to consider the implications of making this change.

Considering the impact of the change

Using CSS3 styling in our example raises an important question – which effect works best? We could always go for a classic linear or swing effect, but these have been used to death. We can easily replace it with something a little more original. In our example, we've used `cubic-bezier(0.23, 1, 0.32, 1)`, which is the CSS3 equivalent of the `easeOutQuint` function.

Working out these effects can be time-consuming. Instead, we can use a great tool created by Lea Verou, which is available at <http://www.cubic-bezier.com>.



To see what our chosen effect looks like in action, head over to <http://cubic-bezier.com/#.23,1,.32,1>. The site has an example we can run to see how the effect will work. The great thing about the site is that we can use the graph to fine tune our effect, which is automatically converted into the relevant values that we can transfer into our code.

This opens up further possibilities – we touched on the use of the Bez plugin from <http://github.com/rdallasgray/bez>; this could easily be used here in place of the standard `.css()` method, to provide our animation.



The CSS equivalents for well-known easing functions (such as `easeInQuint`) are all listed at <https://gist.github.com/tzachyrm/cf83adf77246ec938d1b>; we can see them in action at <http://www.easings.net>.

The important thing though, is the change we can see when viewing the CSS within a DOM Inspector:

```
div.arrow_box tooltipv2.css (line 21)
{
    transition-duration: 2s;
    transition-property: all;
    transition-timing-function:
        cubic-bezier(0.23, 1, 0.32,
        1);
}
```

Instead of applying it inline (as shown in the *Converting animations to use CSS3 automatically* section), we can maintain the separation of concerns principle, by keeping CSS styles in the style sheet, and leaving HTML for organizing our web page content.

Falling back on jQuery animations

Till now, we've used CSS styling to create our animation effect. It raises the question of whether we should use this technique for all our animation requirements, or if jQuery effects should be used.

It all boils down to two key points – how complicated is the animation, and do you need to support older browsers? If the answer to either (or both) is yes, then jQuery is likely to win. If, however, you only have a simple animation, or you don't need to support legacy browsers, then using CSS should be given serious consideration.

The great thing about the animations we've used so far is that we can provide the same effect using both methods – that is, CSS and jQuery. A good source for the easing functions in jQuery is <https://github.com/gdsmith/jquery.easing> - this lists all the standard well-known ones available in libraries such as jQuery UI. To prove we can achieve the same effect, let's go ahead and make a quick change to our code, to use the jQuery equivalent of the animations already used. Go through the following steps:

1. We start by editing a copy of our `quickTip` plugin file. Go ahead and dig out a copy of `jquery.quicktipv2.js`, then add the following block of code, immediately after the variable declarations:

```
$.extend($.easing, {
    easeInQuint: function (x, t, b, c, d) {
        return c*(t/=d)*t*t*t*t + b;
},
```

```

        easeOutQuint: function (x, t, b, c, d) {
            return c*((t=t/d-1)*t*t*t*t + 1) + b;
        }
    });
}

```

2. We now need to adjust our animations to make use of the easing functions, so go ahead and modify the `fadeIn` method, as indicated in the following lines of code:

```

data['tooltip'].stop(true, true).fadeIn(600,
    'easeInQuint');
$this.on('mousemove.' + pluginName, internal.reposition);
},

```

3. We can't have `fadeIn` without its sister `fadeOut`(), so we need to change this call as well, as shown next:

```

$this.off('mousemove.' + pluginName, internal.reposition);
data['tooltip'].stop(true, true).fadeOut(400,
    'easeInQuint');
}

```

4. Save the file as `jquery.quicktipv2.easing.js`. Don't forget to alter the original plugin reference in `tooltipv2.html`! We also need to remove the transition styling for `div.arrow_box` in the `tooltipv2.css` file, so go ahead and comment this code out.

At this point, we have in place a working solution using jQuery. If we preview the results in a browser, the tooltip displays as it should. The downside though, is that we lose visibility of the styling we've used, and that if (heaven forbid) JavaScript is disabled in the browser, then the animation won't play.

There is also the important point that jQuery animations are already more resource hungry, which we touched on back in *Chapter 6, Animating in jQuery*. So why would we resort to using jQuery in these instances, when CSS will work? Again, it's all part of being that better developer – it's too easy to resort to using jQuery; it's right to consider all the alternatives first!



If you design a custom easing, and want to use a CSS equivalent – add the link to the jQuery Animate Enhanced plugin we used earlier. This gives the CSS equivalent using Bezier curve values. We can then use the Bez plugin from earlier, or even bezier-easing from <https://github.com/gre/bezier-easing> to add it to back in as a Bezier curve-based animation instead.

Let's change focus now, and move on. We've provided a limited set of options in our plugin so far; what if we wanted to extend it? We could try delving into the code and adjusting it; though in some cases, this may be overkill for our needs. A better option may be to simply encapsulate it as an instance of a new plugin. Let's take a peek and see what is involved.

Extending our plugin

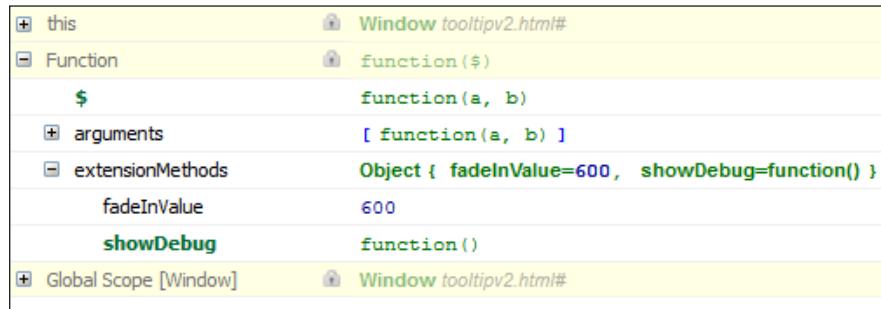
A common problem when using plugins is finding one that meets our requirements completely; the likelihood of that happening is probably less than winning the lottery!

To get around this, we can always extend our plugin, to incorporate extra functionality without affecting existing methods. The benefit of doing this means that we can either override existing methods, or merge in additional functionality that helps mold the plugin towards being a closer fit for our requirements. To see how this would work in action, we're going to add a method and extra variable to our existing plugin. There are lots of ways to achieve this, but the method I've used works well too. Let's go through the following steps:

1. We'll start by editing a copy of `tooltipv2.js`. Immediately below the `#getValue` click handler, go ahead and add the following code:

```
(function($) {
    var extensionMethods = {
        fadeInValue: 600,
        showDebug: function(event) {
            console.log("This is a test");
        }
    }
    $.extend($.fn.quicktip, extensionMethods);
})(jQuery);
```

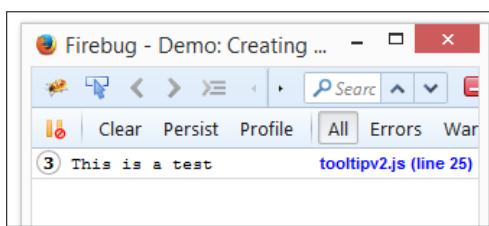
2. Save the file. If we preview `tooltipsv2.html` in a browser, then dig into the rendered code via a DOM Inspector, we should see something akin to the following screenshot:



In this instance, we've added a method that doesn't really perform much; the key here is not so much what it does, but *how we add it in*. Here, we've made it available as an additional method to the existing object. Add the following to the foot of `tooltipsv2.js`:

```
$('#img-list li a.tooltips').on("mouseover", function() {
    $('#img-list li a.tooltips').quicktip.showDebug();
})
```

If we now refresh our browser session, we can see it in action within the **Console** area of our browser, as can be seen in the next screenshot:



There is a lot more that we can do and it's worth spending time researching online. The key to extending is to make sure you understand the differences between `$.fn.extend` and `$.extend`. They might look identical, but trust me – they act differently!

Packaging our plugin using Bower

Okay – on that note, we now have a working plugin and it's ready for use.

At this point, we could just release it as it is, but the smart alternative is to package it for use with managers such as Bower or NPM. This has the advantage of downloading and installing all the required packages, without the need to browse to individual sites and manually download each version.



We can even go to the extent of automating our development workflow with build tools such as Gulp and Grunt – for an example of how, head over to <https://www.codementor.io/bower/tutorial/beginner-tutorial-getting-started-bower-package-manager>.

For now, let's take a quick look at the steps to automate the creation of our Bower package:

1. For this demo, we will need to install NodeJS. So head over to <http://nodejs.org/>, download the appropriate binary or package and install, accepting all defaults.

2. Next, we need to install Bower. Fire up the NodeJS command prompt that will have been installed, and enter the following at the command line:

```
npm install -g bower
```

3. Bower will prompt us for information about the plugin through a series of questions, before displaying the `bower.json` file that it will create for us. In this instance, I've used the tooltip plugin as a basis for our example. The same questions will apply for any plugin you create and want to distribute using Bower, as shown in this screenshot:

The screenshot shows a terminal window with the following text:

```
G:\bower>bower init
[?] name: jquery-quicktip
[?] version: 0.0.1
[?] description: A simple jQuery plugin to display tooltips on elements in a web
[?] description: A simple jQuery plugin to display tooltips on elements in a web
site
[?] main file: js/jquery.quicktip.js
[?] what types of modules does this package expose?
[?] keywords: tooltip
[?] authors: alibby251
[?] license: MIT
[?] homepage: http://www.alexlibby.net
[?] set currently installed components as dependencies? Yes
[?] add commonly ignored files to ignore list? No
[?] would you like to mark this package as private which prevents it from being
[?] would you like to mark this package as private which prevents it from being
accidentally published to the registry? Yes

{
  "name": "jquery-quicktip",
  "version": "0.0.1",
  "authors": [
    "alibby251"
  ],
  "description": "A simple jQuery plugin to display tooltips on elements in a websi
te",
  "main": "js/jquery.quicktip.js",
  "keywords": [
    "tooltip"
  ],
  "license": "MIT",
  "homepage": "http://www.alexlibby.net",
  "private": true
}

[?] Looks good? Yes
G:\bower>
```

4. The final step, after confirming that we are OK with the `bower.json` file that is created, is to register the plugin in Bower. At the command prompt, run the following command:

```
bower register <name of plugin>
```

5. Bower will run through a number of stages before finally confirming that the plugin is available for use via Bower.

At this point, we will have a plugin available for anyone to download. As it has to be linked in with a valid GitHub account, we can now upload the plugin to such an account, and make it available for anyone to download via Bower. As a bonus, we can now take advantage of NodeJS and Grunt to help automate the whole process. How about taking a look at `grunt-bump` (<https://github.com/vojtajina/grunt-bump>), as a starting point?



There is so much more to Bower than we've been able to cover here. For inspiration, it's worth reading the documentation at <http://bower.io/>.



Automating the provision of documentation

The final stage in developing our plugin skills is the provision of documentation. Any coder can produce documentation, but the mark of a better developer is to produce quality documentation, without the need to have to spend lots of time on it.

Enter JSDoc! It is available from <https://github.com/jsdoc3/jsdoc>. If you're not already familiar with it, this is a great way to create your documentation that not only looks good, but can easily be automated using Node. Let's take a moment to install it, and see it work in action. Following steps need to be performed for this:

1. We'll start this time by installing JSDoc via NodeJS. For this we need to bring up a NodeJS command prompt; there will be an icon for this in your **Programs** menu, or from the **Start** page if using Windows 8.
2. At the command prompt, change the location to your project folder, then enter the following command:
`npm install -g jsdoc`
3. Node will run through the installation before confirming that it has completed the process.

To produce documentation, all that needs to happen is for comments to be entered into our code, thus:

```
/**  
 * @name show()  
 * Show the tooltip on screen.  
 * @param event - the name of the event triggering the tooltip to be displayed.  
 */  
show: function(event) {  
    if (debug) console.log(pluginName + '.show()');  
    var $this = $(this), data = $this.data(pluginName);  
  
    data['tooltip'].stop(true, true).fadeIn(600);  
    $this.on('mousemove.' + pluginName, internal.reposition);  
},
```

Once added, the documentation can be compiled by running the following command from within the plugin folder:

```
jsdoc <name of plugin>
```

We will see a folder called out appear; this contains the documentation that we can build up gradually. If we make changes to the comments inline, we need to rerun the compilation process again. This can be automated using the grunt-contrib-watch plugin for Node. If we take a look in the out folder, we can see the documentation appear. It will look similar to the following screenshot extract:

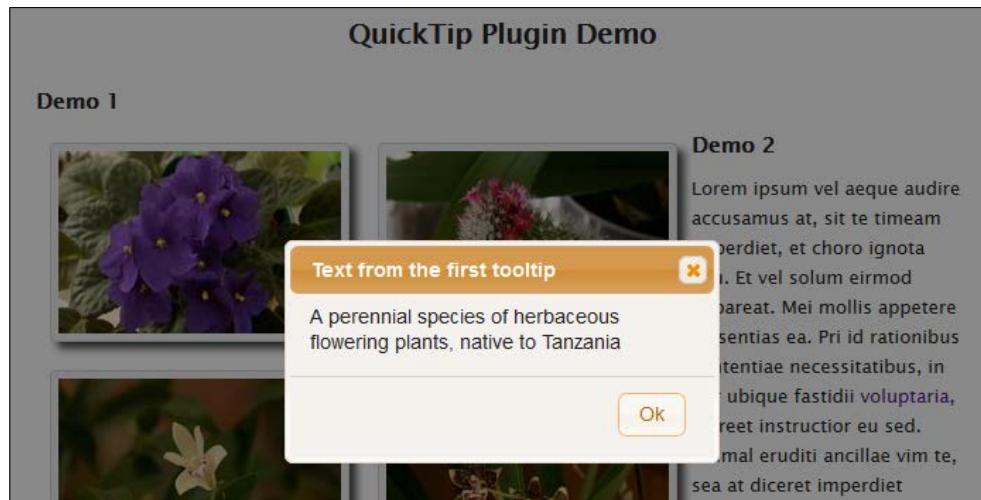


There is a lot more we can cover to get a feel for some of the parameters that can be used to dictate how the documentation will appear, it's worth reading through the extensive documentation at <http://usejsdoc.org/about-getting-started.html>. There are a lot of possibilities available!

Returning values from our plugin

A key part of creating a plugin is – what information can we get back from the plugin? Sometimes we can't get information out, but that may just be a limitation of what we're trying to achieve with the plugin. In our case, we should be able to get the content out. Let's take a look at how we can achieve this with our quicktip plugin.

Before delving into the code, we'll take a look at what we're creating:



1. We need to start somewhere, and so there is no better place than the markup. In a copy of `tooltipv2.html`, go ahead and add the following highlighted code before closing the `<div>` tag:

```
<input type="submit" id="getValue" value="Get text from
first tooltip" />
<div id="dialog" title="Basic dialog">
</div>
```

2. In a copy of `tooltipv2.js`, we need to expose the `data-` tags that we're implementing in the markup. Go ahead and add the configuration option for `tiptag`, as indicated next:

```
$(document).ready(function() {
    $('#img-list li a.tooltips').quicktip({
        class: 'arrow_box', tiptag: "title"
    });
});
```

3. The last step for this part is to modify our markup. In place of using the standard `title=""` tags, we're going to replace them with the `data-` tags, which allow more flexibility. In a copy of `tooltipv2.html`, do a search for all instances of `title`, and then replace them with `data-title`.

4. Next, we need to add in a link to the jQuery UI CSS style sheet. This is purely for creating a dialog box to display the results of us fetching the text from one of the tooltips:

```
<link rel="stylesheet" type="text/css"  
href="http://code.jquery.com/ui/1.10.4/themes/humanity/jquery-ui.  
css">  
<link rel="stylesheet" type="text/css" href="css/tooltipv2.css">
```

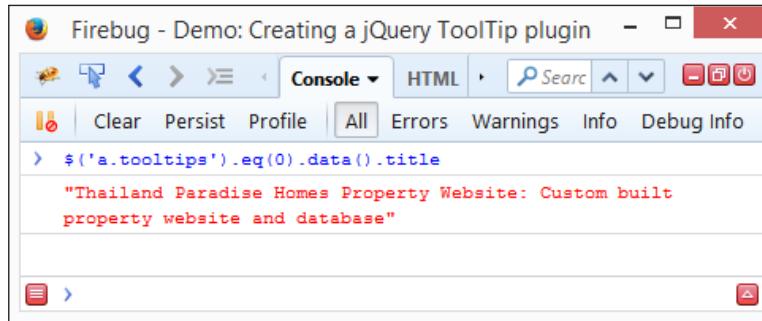
5. To make the jQuery UI CSS work, we need to add a reference to the jQuery UI library. So go ahead and add one in. For convenience, we will use the CDN link, but would look to produce a customized minified version for production use:

```
<script src="http://code.jquery.com/ui/1.11.3/jquery-ui.js">  
</script>  
<script src="js/jquery.quicktipv2.data.js"></script>
```

6. In a copy of tooltip.js, remove all the code within, and replace it with the following:

```
$(document).ready(function() {  
    $('#img-list li a.tooltips').quicktip({  
        class: 'arrow_box',  
        tiptag: "title"  
    });  
  
    $('#getValue').on("click", function(event) {  
        var tipText = $('a.tooltips').eq(0).data().title;  
        $("#dialog").text(tipText);  
        $("#dialog").dialog({  
            title: "Text from the first tooltip",  
            modal: true,  
            buttons: {  
                Ok: function() { $(this).dialog("close"); }  
            }  
        });  
    });  
});
```

7. Save all the files. If we switch to a DOM Inspector such as Firebug, we can see the text returned by entering the highlighted line of code in step 6:



8. In the same browser session, click on the **Get text from first tooltip** button. If all is well, we should see a gentle overlay effect appear, followed by a dialog box with the text displayed, as shown at the start of this exercise.

Granted, our example is a little contrived, and that we should look to not hardcode in the reliance on fetching the text from the first tooltip, but by selecting text from whichever tooltip we desired. The key though is that we can equally customize both the tags used for the text, and also retrieve that content using the `.data()` method, with ease.

Exploring best practices and principles

Over the last few pages, we've covered a number of concepts and tips that we can use to help develop our plugin skills further. There are a few additional factors worth considering, which we've not covered yet. It's worth taking a few moments to explore these factors:

- **Quality and Code Style:** Have you considered linting your plugin code through JSHint (<http://www.jshint.com>), or JSLint (<http://www.jslint.com>)? Adhering to best practices when writing jQuery is one way to help ensure success, such as following a consistent code style or the guidelines issued at <http://contribute.jquery.org/style-guide/js/>? If not, how clean and readable is your code?
- **Compatibility:** Which version of jQuery is your plugin compatible with? Significant changes have been made to the library over the years. Are you intending to provide support to older browsers (requiring the 1.x branch of jQuery), or staying with more modern browsers (using version 2.x of the library)?

- **Reliability:** You should consider providing a set of unit tests. These help prove that the plugin works, and are easy to produce. For a guide on how to do this with QUnit, you may like to take a look at *Instant Testing with QUnit*, by Dmitry Sheiko, available from Packt Publishing.
- **Performance:** A plugin that is slow to run will turn potential users away. Consider using JSPerf.com (<http://www.jsperf.com>) to test segments as a benchmark for assessing how well the plugin works, and whether any section needs further optimization.
- **Documentation:** Documenting your plugin is a must. The level of documentation will often determine the plugin's success or failure. Does the plugin contain any quirks that developers need to be aware of? What methods and options does it support? It will also help if the code has inline comments, although it helps to provide a minified version for production use. If a developer can navigate your code base well enough to use it or improve it, then you've done a good job.
- **Maintenance:** If we release something into the wild, then thought must be given to a support mechanism. How much time do we need to offer maintenance and support? It is critical to be clear upfront about what expectations are around answering questions, addressing issues, and continuing to improve the code.

Phew – there's a lot to consider! Creating a plugin can be a rewarding experience. Hopefully, some of these tips will help improve your skills and make you a more rounded developer as a whole. Remember, anyone can write code, as I always say. The key to becoming a better developer is understanding what makes a good plugin, and knowing how to put that into practice.



The Learn jQuery site has a few extra tips that are worth exploring at <http://learn.jquery.com/plugins/advanced-plugin-concepts/>.

Summary

If someone asked you the name of one topic that is key to learning jQuery – it's very likely that plugins would feature highly in that answer! To help with writing them, we've covered a number of tips and tricks in this chapter. Let's take five to recap what we've learnt.

Our starting point was a discussion on detecting the signs of poorly developed plugins, as a precursor to learning about how we can improve our development through the use of plugin patterns. We then moved on to working through the design and construction of an advanced plugin, starting with creating the basic version before reordering it to use a boilerplate template.

Next came a detailed look at switching to using CSS3 animations, to develop some of the arguments we covered earlier in the book, in considering the use of CSS3 to better manage animations than resorting to jQuery.

We then moved on to looking at how we can extend functionality in our plugin, before learning about packaging it with Bower ready for use through GitHub. We then covered the automatic provision of documentation, and how we can return values from our plugin, before rounding up with a look at some of the best practices and principles we can take away for use in our development.

Right – onwards we go! In the next chapter, we're going to mix jQuery (including some plugins), HTML5 markup, and CSS, and produce a site. Okay, nothing outrageous with that – that's perfectly normal. Here comes the twist though: what about running the site in its entirety, *offline*? Yes, you heard me right...offline...and no, there's not a USB key or DVD in sight either...

12

Using jQuery with the Node-WebKit Project

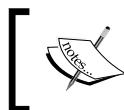
In this modern age, responsive design is the latest buzzword, where websites built using jQuery can work correctly on any device or platform. Nevertheless, this requires an Internet connection—what if we can develop an offline version of the same app?

Enter Node-WebKit (or NW.js, as it is now known). In this chapter, we're going to take a break from exploring jQuery and explore one of the lesser-known ways of using the library instead. You'll see how you can use the power of jQuery, HTML5, and the desktop, mixing them to produce a replica of your site that works offline in any desktop or laptop environment. We'll use it to have a little fun with developing a simple file size viewer that uses jQuery, which can be easily developed into something more complex that can run online or offline, as needed.

In this chapter, we'll cover the following topics:

- Introducing Node-WebKit
- Building a simple site
- Packaging and deploying your app
- Taking things further

Ready to explore the world of Node-WebKit? Let's make a start...



You may see references to NW.js online—this is the new name for Node-WebKit, as of January 2015; you may see both names being used throughout this chapter.

Setting the scene

Imagine a scene, if you will, where a client has asked you to produce a web-based application; they've outlined a specific set of requirements, as follows:

- It must have a simple GUI
- There shouldn't be any duplicates – it must be one version that works on all platforms
- The solution must be easy to install and run
- It needs to be portable so that it can be transferred if we change computers

Hands up if you think a website will suffice? Now, hands up if you haven't read the requirements properly...!

In this instance, a website isn't going to be enough; a desktop application will deal with the duplication requirement, but it may not be easy to use and certainly won't be cross-platform. So, where do we go from here?

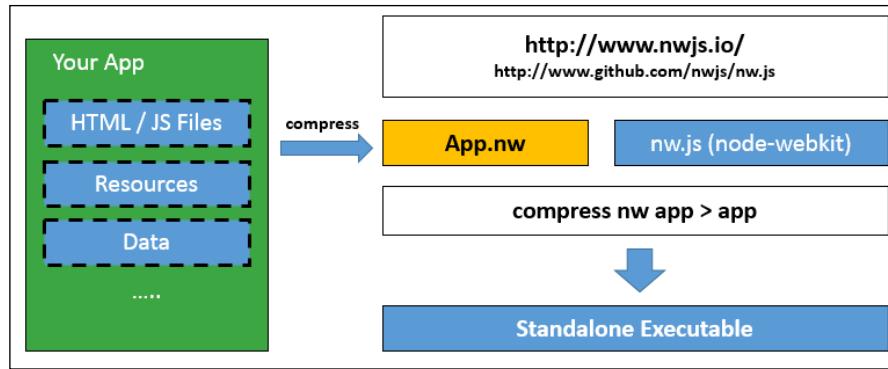
Introducing Node-WebKit

Node-WebKit (or NW.js, as it is now known) was originally created by Intel but open sourced in 2011 and is available at <http://nwjs.io/>; the project is an attempt to combine the best of SPA development with an offline environment (where hosting a web server is not practical).

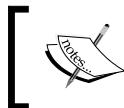
Node-WebKit is based on Chromium, a WebKit-based browser that has been extended in order to allow you to control user interface elements that are normally off-limits to web developers. The security model has been relaxed (on the basis that the code we're running is trusted) and that it integrates NodeJS; this opens up an array of possibilities, outside of what would normally be possible with HTML5 APIs.

At first, it may seem like a complicated mix. However, fear not as most finished solutions built in nothing more than plain HTML, CSS, and JavaScript, with a sprinkling of images to finish it off.

The basic principle, as we will see throughout this chapter, is to produce a normal site and then compress HTML, CSS, and all related resource files into one ZIP file. We simply rename it to have an .nw extension and then run the main nw.exe application. Provided that we've set up a requisite package.json file, it will automatically pick up our application and display it on the screen, as shown here:



Hold on though; this book is about jQuery, right? Yes, absolutely; here comes the best part: Node-WebKit allows you to run standard JavaScript and jQuery along with any Node third-party modules! This opens up a wide variety of opportunities; we can use the main library or any of a host of additional jQuery-based libraries, such as Three.js, AngularJS, or Ember.



The only key part that we really have to remember is that there are some quirks of using NW.js, such as using a folder dialog to browse and select local folders; we will cover this in more detail later in this chapter.



At this point, I am sure you will be asking yourself one question: why would I want to use nw.js (or Node-WebKit)? This is a perfectly valid question; it might well seem illogical that we're running a web-based site as a desktop application! In this apparent madness, there are some valid reasons for doing this, so let's take a look at them now and see why it makes sense to run a site as a desktop application.

Operating HTML applications on a desktop

As developers, one of the biggest headaches we face is ensuring that users have the same experience across all the browsers that we need to support when accessing our site. Now, I should make it clear: in terms of the same experience, there may be instances where this simply isn't possible, so we have to at least provide a graceful exit path for those browsers that don't support a particular piece of functionality.

Thankfully, this concern is slowly but surely becoming less of an issue. The great thing about Node-WebKit is that we only have to support Chrome (as this is what Node-WebKit is based on).

In most cases, we can simply reuse the code created for Chrome; this allows us to easily push out cross-platform applications using frontend frameworks (including jQuery!) and Node modules that we already know or use. In addition to this, there are several reasons why you will use Node-WebKit to help produce cross-platform applications, as follows:

- Access to the latest web technologies available in Blink, the rendering engine behind Google Chrome.
- NW.js supports the *build once, run anywhere* concept – this may not suit all applications, but many will benefit from sharing code between the desktop, web, and mobile environments.
- If you want your app to run at a certain size or do some more advanced things with popups, you get this control on the desktop. Most solutions also provide a way to access the file system and allow other more advanced controls that you wouldn't get with a regular web application.

Without wanting to appear negative, there are some considerations that you need to be aware of; the principal concern is the size of the executable.

A site or an application created with native UI libraries, such as jQuery, may only be a few kilobytes in size. An equivalent version built using Node-WebKit will be significantly bigger, as it includes a cut-down version of Node and Chromium. It's for this reason that you need to be careful about file sizes – you can use some of the tips and tricks from *Chapter 2, Customizing jQuery*, to reduce the size of jQuery. There are a couple of other concerns that you need to be mindful of; they include the following:

- Compared to native applications, desktop web applications typically require a much larger amount of RAM and CPU power to run and render.
- In terms of appearance, if you want to make your application look good on the platform you're planning on deploying to, then you'll need to either recreate common UI elements using CSS or create a totally new UI, including a new design for every operating system-provided UI element, such as the title bar, menu bar, and context menus.

- Although Node-WebKit relaxes some of the security issues that are otherwise found when using browser applications (such as the same origin policy), you still only have access to the Node-WebKit context; and in some instances, you have to use WebKit-specific tags, such as `nwdirectory`, when creating a select directory dialog. The net effect means an increase in code, if you want to create one file that supports both web and desktop environments. You can mitigate against the effects of this issue: <http://videlais.com/2014/08/23/lessons-learned-from-detecting-node-webkit/> provides a useful trick to determine which environment you are in and allows you to reference the appropriate files needed for that environment.



For more information on some of the security considerations, take a look at the security page on the NW.js Wiki, available at <https://github.com/nwjs/nw.js/wiki/Security>.

Now that we've been introduced, let's delve in and get started with installing Node before we start building our jQuery-based application. It should be noted that the focus of this chapter will be largely based on Windows, as this is the platform that is used by the author; changes will need to be made for those using Linux or Mac platforms.

Preparing our development environment

Over the next few pages, we're going to build a simple application that displays the file sizes of any files dropped into the main window or selected via a file dialog. In reality, we wouldn't use the application on its own, but as a basis for uploading images for processing or perhaps as the offline version of a compression application. There are plenty of ways in which we can develop it further—we will touch on some ideas later in the chapter, in the *Taking things further* section.

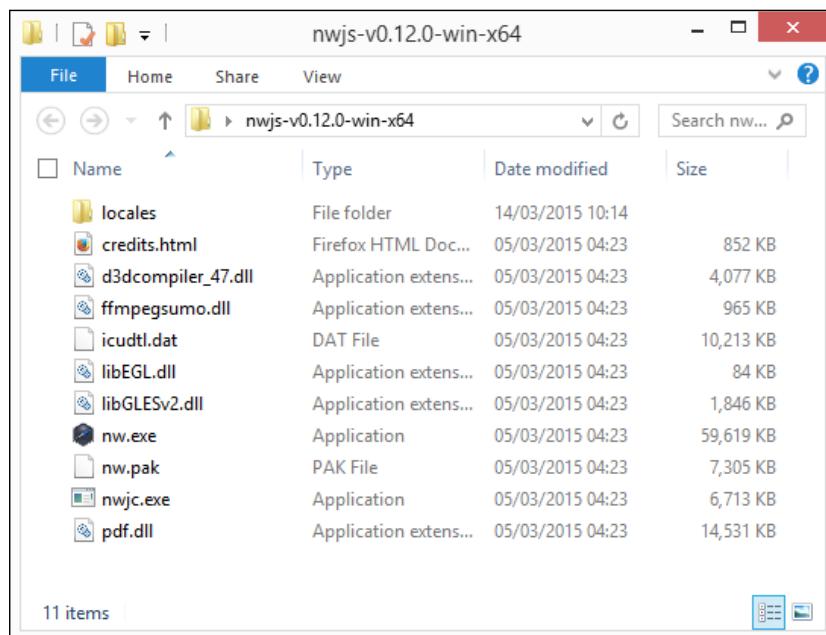
In the meantime, let's get started with installing NW.js. Before doing this, we need to avail ourselves of the following tools:

- A compression program is needed; on the Windows platform, you can use the in-built capabilities or something such as 7-Zip (<http://www.7-zip.org>), if preferred.
- We will need a text editor; throughout the course of this chapter, we will use Sublime 2 or 3, but any good text editor should suffice if you already have a personal preference. Sublime Text can be downloaded from <http://www.sublimetext.com>, with versions available for the Mac, Linux, and Windows platforms.

- We'll be making use of Node and Grunt to install additional packages. Node is available at <http://www.nodejs.org>, so go ahead and install the version suitable for your platform. Once installed, run this command from a NodeJS command prompt to install Grunt:

```
npm install -g grunt-cli
```

- Last, but by no means least, we need the Node-WebKit library (of course), so head over to <http://nwjs.io/> and download the version appropriate for your platform. If you expand the folder, you should see something similar to what is shown in this screenshot:



As an aside, Node-WebKit can be easily integrated into existing Grunt files, which means that we can take the advantage of packages such as `cssmin` to minify the CSS style sheets we create for our application. It is definitely worth exploring as you become more familiar with Node-WebKit.

Enough of the chit-chat; it's time for us to start developing! As with all other things, we need to start somewhere. Let's have a crack at creating a simple "Hello World" example, before we look at how to use jQuery.

Installing and building our first application

I wonder: how many times have you read books or online articles about a programming language, which provide their own take on the ubiquitous "Hello World" example? I'll bet it must be quite a few times over the years...and yes, before you ask, we're not going to break the tradition either! Following in the footsteps of anyone who has provided "Hello World" examples, here's our own take.



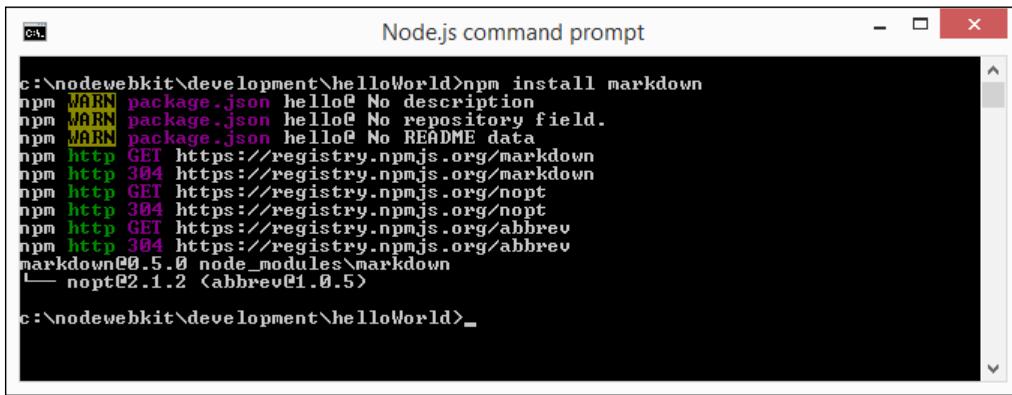
To build this, we need to do the following:

1. Browse to <http://nwjs.io/> and download the package for your platform; we will assume the use of Windows for now, but packages are available for Mac and Linux platforms as well.
2. Extract the node-webkit-vX.XX.XX-win-x64 folder (where XX is the version number), rename it as nodewebkit, and copy it to your main PC drive—Linux or Mac users can copy this folder to their user areas. Once done, create a new folder called development within the nodewebkit folder.
3. Next up, we need to install NodeJS. To do this, head over to <http://nodejs.org/download/> in order to download and install a version suitable for your platform, accepting all the defaults.

Node-WebKit can use any of the standard Node packages available. As an example, we're going to install the `markdown` package, which converts suitably marked up plain text to valid HTML. Let's continue the exercise by installing it and seeing how it works:

1. In the NodeJS command prompt, change to the `helloworld` folder and then enter the following code and press *Enter*:

```
npm install markdown
```



```
c:\nodewebkit\development\helloworld>npm install markdown
npm WARN package.json hello@ No description
npm WARN package.json hello@ No repository field.
npm WARN package.json hello@ No README data
npm http GET https://registry.npmjs.org/markdown
npm http 304 https://registry.npmjs.org/markdown
npm http GET https://registry.npmjs.org/nopt
npm http 304 https://registry.npmjs.org/nopt
npm http GET https://registry.npmjs.org/abbrev
npm http 304 https://registry.npmjs.org/abbrev
markdown@0.5.0 node_modules\markdown
└── nopt@2.1.2 (abbrev@1.0.5)

c:\nodewebkit\development\helloworld>
```

2. Close the window as you don't need it. Next, extract a copy of the `index.html` and `package.json` files from the `helloworld` folder in the code download that accompanies this book; save these in the `helloworld` folder in your project area.
3. Create a new ZIP folder called `helloworld.zip` and then add these two files to it; rename `helloworld.zip` to `helloworld.nw`.

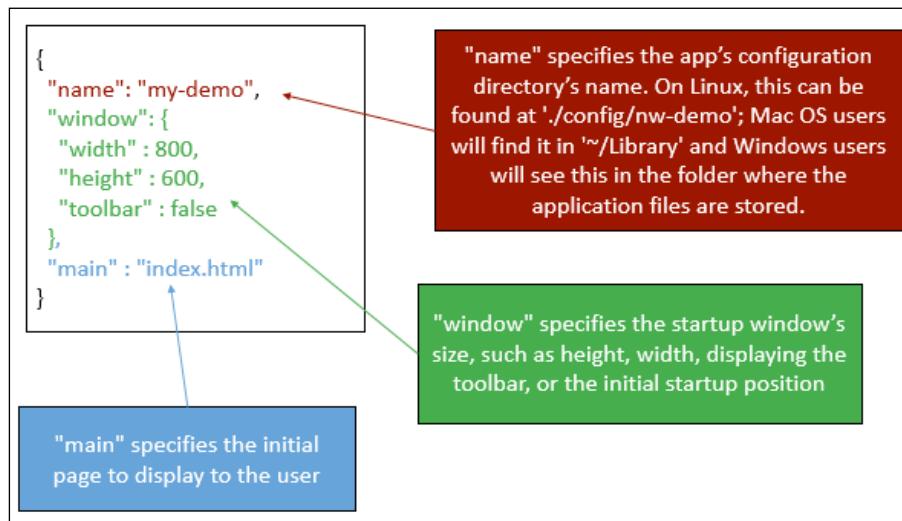
We can now run our application; there are three ways to do this with Node-WebKit:

- In the NodeJS command prompt, switch to the `nodewebkit` folder and then run the following command:
`nw C:\nodewebkit\development\helloworld.nw`
- Double-click on the `nw.exe` application; this will pick up the `package.json` file and run the `helloworld.nw` file automatically
- Drag and drop the `helloworld.nw` file onto `nw.exe` to run the application

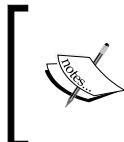
Whichever route you prefer to use, running it will show the **Hello World** window shown at the start of this exercise. It's a simple, no-frills example of using Node-WebKit—granted it won't win any awards, but it shows how simple it is to create a functional application from existing HTML pages.

Dissecting the package.json file

At the heart of our application is the `package.json` file. This manifest file tells Node-WebKit how to open the application and controls how the browser should behave:



It's worth getting to know this file in detail; it holds all the metadata for the project and follows the standard format for all Node-based packages. If you're not familiar with the manifest file, you can see a detailed example at <http://browsenpm.org/package.json> with interactive explanations for each section; Node-WebKit's version works in a similar fashion.

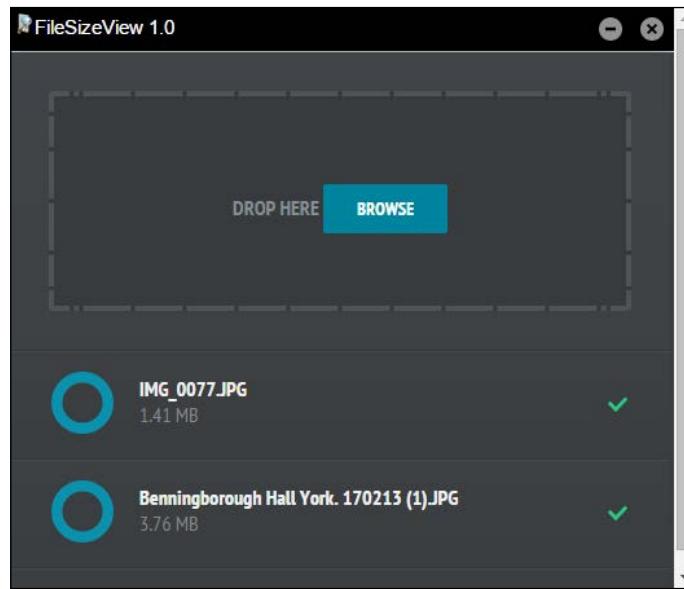


For more in-depth details about the Node-WebKit manifest file and the components that make it up, head over to the documentation on the main NW.js site (<https://github.com/nwjs/nw.js/wiki/manifest-format>).

Right, it's time to get stuck in and build our example application!

Building our simple application

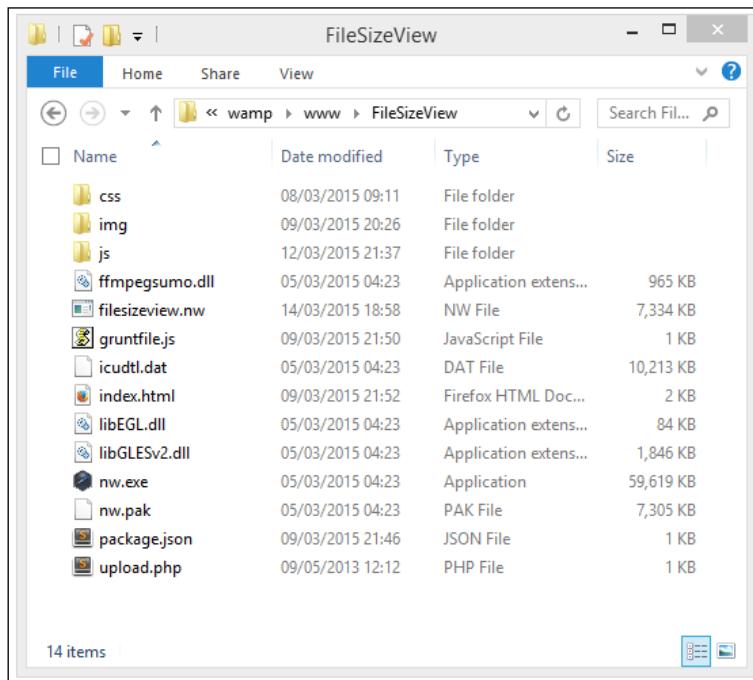
Over the next few pages, we're going to build a simple application that allows us to drag and drop a file onto a drop zone in order to render the file sizes. It's based on the tutorial by Martin Angelov, which is available at <http://tutorialzine.com/2013/05/mini-ajax-file-upload-form/>; we'll concentrate on the frontend UI interface and not worry about the backend upload facility for the purposes of our demo:



Even when just working on the frontend user interface, there's still a fair amount of code involved; our focus will be primarily on the jQuery code, so let's take a look at the demo in action first before exploring it in more detail. To do this, perform the following steps:

1. We use a small bit of PHP code in our demo, so we need to set up web server space first, such as WAMP (for a PC – <http://www.wampserver.de/en>) or XAMPP (or MAMP for Mac – <http://www.mamp.info/en>). Linux users will have something available from within their distro. We'll use WAMP for this demo – please adjust locations accordingly if yours are different; use the default settings when installing it. If you prefer a cross-browser solution, then XAMPP is a good option – it's available at <https://www.apachefriends.org/index.html>.

2. Next up, we need to extract a copy of the `FileSizeView` folder from the code download that accompanies this book. This contains the markup required for our application to work. Save the folder within `C:\wamp\www`.
3. We need a copy of Node-WebKit to run our application, so go ahead and copy the contents of the `nwjs` folder that is in the code download into the `FileSizeView` folder. If all is well, you should have the files shown as follows:



4. At this stage, if we double-click on `nw.exe`, we should see our application run. Also, you will see the window displayed at the start of this exercise.

Okay, so it shows the window; "how does it all work," I hear you ask? Well, there are a few key points to note from this exercise, so let's spend some time to go through things in more detail.

Exploring our demo further

If you take a look at the `FileSizeView` folder in more detail, you should see that most of the content centers around the `index.html` and `upload.php` files, with the associated CSS, image, and JavaScript files needed to make the demo work. In addition, we have a number of files from the Node-WebKit folder – these provide a cut-down version of Node and Chromium, which is used to host our files:

- `nw.exe` and `nw.pak`: This is the main Node-WebKit executable and JavaScript library file that runs our code, respectively.
- `package.json`: This is a manifest file that we saw in use earlier in the chapter, in the *Installing and building our first application* section; this provides directions to Node-WebKit on how to display our application.
- `ffmpegsumo.dll`: This is used to provide video and audio support; it isn't necessary for our demo but can be included for future use.
- `filesizeview.nw`: This is our zipped up application; this is the file that Node-WebKit runs once it has checked `package.json` to verify how it should be displayed.
- `gruntfile.js`: This is the Grunt file for `grunt-node-webkit-builder`, which we will use later in *Automating the process* to compile our files into one application.
- `icudtl.dll`: This is a network library required by Node-WebKit.
- `libEGL.dll` and `libGLESv2.dll`: These files are used for **Web Graphics Library (WebGL)** and GPU acceleration.

In some Node-WebKit applications that are available online, you may see the presence of `D3DCompiler_43.dll` and `d3dx9_43.dll` too. These are from the DirectX redistributable and are used to provide increased WebGL support.

Dissecting our content files

Okay, so we have our main Node-WebKit files; what else are we using? Well, in addition to the standard HTML markup, images, and styles, we also use a number of jQuery-based plugins and some custom jQuery code to tie together.

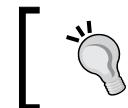
The main plugin files in use are jQuery, jQuery UI, jQuery Knob, and the BlueImp file upload plugin. We also use some custom code to tie this all together – they are in `window.js` and `script.js`. Let's take a look at these in more detail, beginning with `window.js`.

Exploring window.js

In `window.js`, we first make a call to `nw.gui`, the native UI library for Node-WebKit that uses `require()`; this is a standard format for calling any module, such as internal ones or even external third-party modules. We then assign this to the `gui` variable before using this to get a handle on the window of our application:

```
var gui = require('nw.gui'), win = gui.Window.get();
```

Note that as we can only access the Node-WebKit context, we must use the dedicated library; we cannot access the window using a standard JavaScript call.



For more information on accessing modules, take a look at the documentation available at <https://github.com/nwjs/nw.js/wiki/Using-Node-modules>.



Next up, we set two delegated document handlers, one to handle the minimizing of the window and the other to close it completely:

```
$(document).on('click', '#minimize', function () {
    win.minimize();
});

$(document).on('click', '#close', function () {
    win.close();
});
```

This scratches the surface of what we can do; there is so much more. Head over to <https://github.com/nwjs/nw.js/wiki/Window> in order to get a feel of what is possible to achieve.

Dissecting the BlueImp plugin configuration

The main functionality within our site is hosted in `script.js`. This contains the main configuration object for the BlueImp file upload plugin along with some additional helpers. Let's take a look at it in more detail.

We start with the normal document-ready call before assigning a reference to the `#upload li` list item as a variable, as shown here:

```
$(function(){
    var ul = $('#upload ul');

    $('#drop a').click(function(){
        // Simulate a click on the file input button
```

```
// to show the file browser dialog
$(this).parent().find('input').click();
});
```

Next up, we configure the file upload plugin. First, we set the initial drop zone to the #drop selector:

```
// Initialize the jQuery File Upload plugin
$('#upload').fileupload({

    // This element will accept file drag/drop uploading
    dropZone: $('#drop'),
```

We then set up the add callback function. This deals with displaying each list item that has been added to the list, either via drag and drop or by browsing for the file. We start by creating a template and then cache it in the tpl variable:

```
add: function (e, data) {
    var tpl = $(<li class="working"><input type="text" value="0"
        data-width="48" data-height="48" + ' data-fgColor="#0788a5"
        data-readOnly="1" data-
        bgColor="#3e4043"/><p></p><span></span></li>');
```

We then find the filename that has just been added, before working out and appending the filesize function to the list:

```
tpl.find('p').text(data.files[0].name).append('<i>' +
    formatFileSize(data.files[0].size) + '</i>');

    // Add the HTML to the UL element
    data.context = tpl.appendTo(ul);
```

Next up, we initialize the jQuery Knob plugin. Although it is nonoperational for now, it will produce a good circular status gauge of the progress in uploading any file to the remote location:

```
// Initialize the knob plugin
tpl.find('input').knob();
```

At the moment, we're not using the cancel icon. This will be the event handler we'd need to use to work out if we cancel the upload of any item while it is in progress:

```
tpl.find('span').click(function(){

    if(tpl.hasClass('working')){
        jqXHR.abort();
```

```
        }

        tpl.fadeOut(function() {
            tpl.remove();
        });
    });

    // Automatically upload the file once it is added to the queue
    var jqXHR = data.submit();
},

```

This is the key method handler within the `fileupload` object. This takes care of working out the percentage value of progress in uploading the file before triggering a change to update the jQuery Knob plugin, as shown here:

```
progress: function(e, data){
    var progress = parseInt(data.loaded / data.total * 100, 10);
    data.context.find('input').val(progress).change();
    if(progress == 100){
        data.context.removeClass('working');
    }
},

```

If the file fails to upload, then we set a class of `.error`, which is appropriately styled within the accompanying style sheet:

```
fail:function(e, data) {
    // Something has gone wrong!
    data.context.addClass('error');
}
);

```

In addition to the main `fileupload` configuration object, we also set a couple of helper functions. The first helper function prevents the normal action that should take place if we drag anything over the document object, which will be an attempt to display it within the browser window:

```
$(document).on('drop dragover', function (e) {
    e.preventDefault();
});

```

The second helper function handles the conversion of the file size from a byte value to either its kilobyte, megabyte, or gigabyte equivalent, before returning the value for rendering on the screen:

```
function formatFileSize(bytes) {
  if (typeof bytes !== 'number') {
    return '';
  }

  if (bytes >= 1000000000) {
    return (bytes / 1000000000).toFixed(2) + ' GB';
  }

  if (bytes >= 1000000) {
    return (bytes / 1000000).toFixed(2) + ' MB';
  }

  return (bytes / 1000).toFixed(2) + ' KB';
}
);
```

At present, there is definitely scope for improvement in our project: it will work fine within a normal browser window but needs modification to make it operate 100 percent properly within a Node-WebKit context. We'll cover some ideas as to where we can improve the code later, within the *Taking things further* section, but for now, there is one important tip we need to cover off before we consider debugging our application.

Automating the creation of our project

One key theme that I've tried to maintain throughout this book is how we can be smarter at doing things; anyone can write code, but the smarter developer knows when it is time to automate some of the more mundane tasks and use their time on tasks that will return more value.

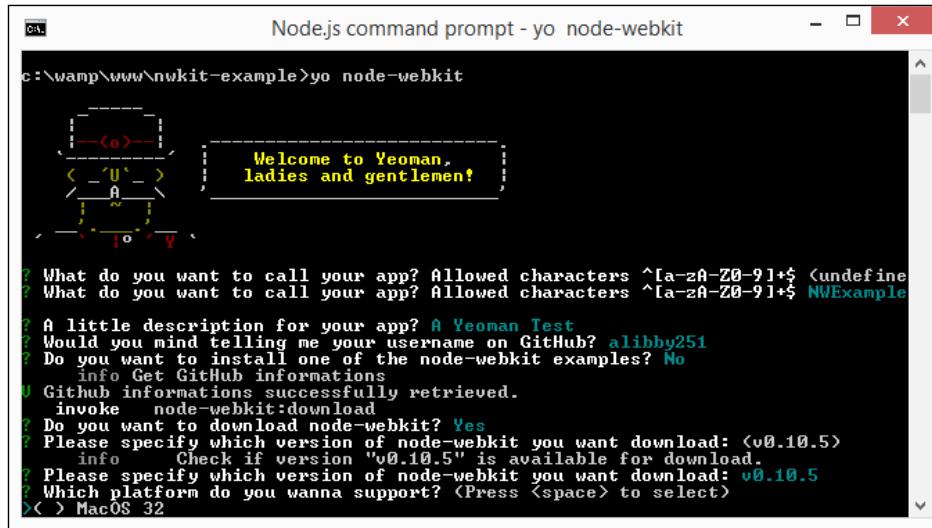
One way in which we can improve on creating and building our project is to automate the generation of our skeleton project. Thankfully, we can do this using the Yeoman generator for node-webkit applications (available at <https://github.com/Dica-Developer/generator-node-webkit>), which we can install using the following command:

```
npm install -yeoman
```

The preceding command is followed by this:

```
npm install -g generator-node-webkit
```

This displays the following screenshot, which shows the details being entered for a test project:

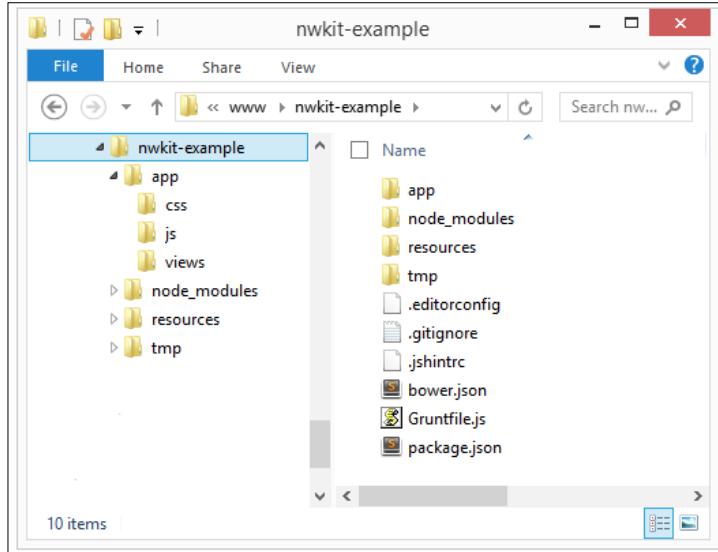


```
Node.js command prompt - yo node-webkit
c:\wamp\www\nwkit-example>yo node-webkit

Welcome to Yeoman,
ladies and gentlemen!

? What do you want to call your app? Allowed characters ^[a-zA-Z0-9]+? <undefined>
? What do you want to call your app? Allowed characters ^[a-zA-Z0-9]+? NWExample
? A little description for your app? A Yeoman Test
? Would you mind telling me your username on GitHub? alibby251
? Do you want to install one of the node-webkit examples? No
  info Get GitHub informations
  0 Github informations successfully retrieved.
  invoke node-webkit:download
? Do you want to download node-webkit? Yes
? Please specify which version of node-webkit you want download: <v0.10.5>
  info Check if version "v0.10.5" is available for download.
? Please specify which version of node-webkit you want download: v0.10.5
? Which platform do you wanna support? <Press <space> to select>
>< > MacOS 32
```

If all went well, you should see your predefined folder structure in place, ready for you to use, as shown in the following screenshot:



This makes it a lot easier to create the folder structure needed and to maintain consistency in projects.

Debugging your application

At this point, you should have a working application that you can deploy. While it has to be said that ours needs more work before it will be ready for release, the principles behind deployment are still the same, irrespective of the application! There is one small thing I want to cover off, before we look at deployment.

Remember how I mentioned Sublime Text will be used throughout this chapter? Well, there's a good reason for this: it lends itself perfectly to build the application to a point where we can run it and debug the application. To do this, we need to create a new build system file for Sublime Text (such as the one outlined as follows, for Windows):

```
{  
  "cmd": ["nw.exe", "--enable-logging",  
          "${project_path}:${file_path}"],  
  "working_dir": "${project_path}:${file_path}",  
  "path": "C:/Tools/nwjs/",  
  "shell": true  
}
```

The process to add in the new build file for Sublime is quick—for full details, head over to <https://github.com/nwjs/nw.js/wiki/Debugging-with-Sublime-Text-2-and-3>. It's a useful trick to use while developing your application, as the manual build process can get very tedious after a while!

Packaging and deploying your app

Okay, so we have a working application that is ready for packaging and deployment; how do we turn it into something that we can make available for download?

Packaging a Node-WebKit application is surprisingly easy. There are a couple of caveats, but in the main the process centers around dropping all the Node-WebKit distributable files into a folder along with our content and shipping it as a renamed zipped file.

There are several different ways to package our files, depending on the platform being used. Let's take a look at a couple of options using the Windows platform, beginning with a manual compilation.



For those of you who work on Apple Macs or Linux, details on how to package apps are available at <https://github.com/rogerwang/node-webkit/wiki/How-to-package-and-distribute-your-apps>.

Creating packages manually

Assuming that we're ready to deploy our application, these are the basic steps to follow when creating packages manually—for this example, we'll use the files created earlier, in the *Building our simple application* section:

1. Create a new blank ZIP file and add the package.json, ffmpegsumo.dll, icudtl.dat, libEGL.dll, libGLESv2.dll, and nw.pak files—these are needed to host the site within the cut-down version of Chromium and Node.
2. Add the css, img, and js folders along with index.html to the ZIP file.
3. Rename ZIP to the .nw file and then run nw.exe—this will use the package.json file to determine what should be run.



Note that Node-WebKit packages do not protect, obfuscate, digitally sign, or make the package secure; this means that making your package open source is a much better option, if only to avoid any problems with licensing!

Automating the process

Hang on, creating a package is a manual process that gets tedious after a while if we're adding a lot of changes, right?

Absolutely, the smart way forward is to automate the process; we can then combine it with a Grunt package, such as grunt-contrib-watch (from <https://github.com/gruntjs/grunt-contrib-watch>), to take care of building our packages as soon as any change is made. There are several ways to automate it—my personal favorite is to use grunt-node-webkit-builder, from <https://github.com/mllrsohn/grunt-node-webkit-builder>.



The node-webkit-builder plugin was created by the same developers as the ones behind grunt-node-webkit-builder; the only difference is that the latter has additional support for use with Grunt. If you want to switch to using Grunt, you can install a supplementary package, grunt-node-webkit-builder-for-nw-updater, which is available at <https://www.npmjs.com/package/grunt-node-webkit-builder-for-nw-updater>.

Let's take a look at the plugin in action—the exercise assumes that you have NodeJS already installed, before continuing with the demo:

1. In a new file within the project folder, add the following code and save it as `gruntfile.js`:

```
module.exports = function(grunt) {  
  
    grunt.initConfig({  
        nodewebkit: {  
            options: {  
                platforms: ['win'],  
                buildDir: './builds',  
                winIco: './img/filesize.ico'  
            },  
            src: ['./css/*.css', './img/*.*', './js/*.js',  
                  '*.html', '*.php', '*.json', '*.ico']  
        }  
    })  
  
    grunt.loadNpmTasks('grunt-node-webkit-builder');  
    grunt.registerTask('default', ['nodewebkit']);  
};
```

2. Next up, we need to install `grunt-node-webkit-builder`; therefore, go ahead and fire up an instance of the NodeJS command prompt and then navigate to the project folder, which we used earlier in the *Building our simple application* section.
3. Enter this command, then press *Enter*, and wait for it to complete:

```
Npm install grunt-node-webkit-builder --save-dev
```

4. In the `package.json` file, you will see that the following lines have been added, as indicated:

```
"icon": "img/filesize.png"  
},  
"devDependencies": {  
    "grunt": "~0.4.5",  
    "grunt-node-webkit-builder": "~1.0.2"  
}
```



If you need to see what the package.json will look like, then head over to <https://github.com/3dd13/sample-nw>. There is a sample file at <https://github.com/3dd13/sample-nw/blob/master/package.json>, which shows the contents of the code we've just entered into our own version of the file.

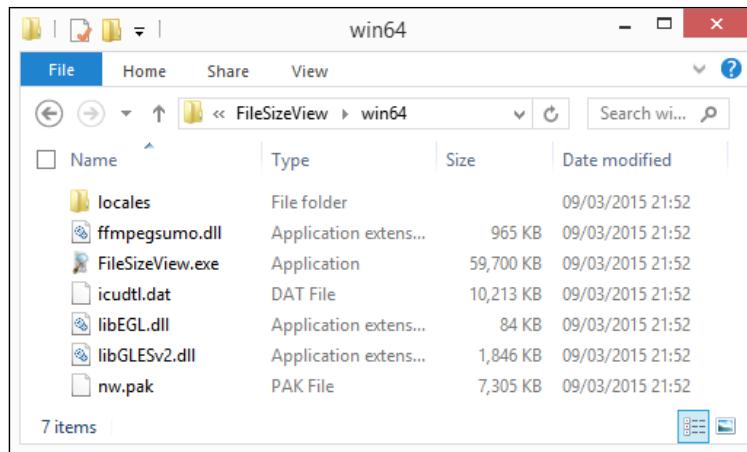
5. At this stage, we're now ready to build our package. At the prompt, type grunt and then wait for it to complete; you should see it build the package, as shown in the following screenshot:

The screenshot shows a Windows command prompt window titled "Node.js command prompt". The command entered was "grunt". The output shows the process of building a NW.js application, including creating cache and release folders, updating executable icons, and zipping various files. The final message indicates the "nodewebkit app created" and "Done, without errors".

```
c:\wamp\www\FileSizeView>grunt
Running "nodewebkit:src" (nodewebkit) task
Latest Version: v0.12.0
Using v0.12.0
Create cache folder in c:\wamp\www\FileSizeView\cache\0.12.0
Using cache for: win32
Create cache folder in c:\wamp\www\FileSizeView\cache\0.12.0
Using cache for: win64
Create release folder in c:\wamp\www\FileSizeView\builds\FileSizeView\win64
Create release folder in c:\wamp\www\FileSizeView\builds\FileSizeView\win32
Update win32 executable icon
Update win64 executable icon
Zipping css/style.css
Zipping img/filesize-small.png
Zipping img/border-image.png
Zipping img/filesize.ico
Zipping img/filesize.png
Zipping img/icons.png
Zipping img/icons.svg
Zipping js/jquery.fileupload.js
Zipping js/jquery.iframe-transport.js
Zipping js/jquery.min.js
Zipping js/jquery.knob.js
Zipping js/jquery.ui.widget.js
Zipping js/main.js
Zipping js/script.js
Zipping js/window.js
Zipping index.html
Zipping upload.php
Zipping package.json
>> nodewebkit app created.

Done, without errors.
c:\wamp\www\FileSizeView>
```

6. If you revert to the folder where our files are stored, you should now see that a `builds` folder has appeared; navigating through it will show you something similar to this screenshot, where you have the contents of the `win64` build folder displayed:



At this stage, we can double-click on the `FileSizeView.exe` application to launch the program. This will display our application in all its glory, ready for use. Perfect! We can deploy the files now, right?

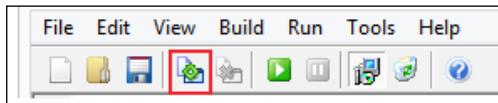
Deploying your application

Mmm...hold your horses; as you should know by now, we can always do better!

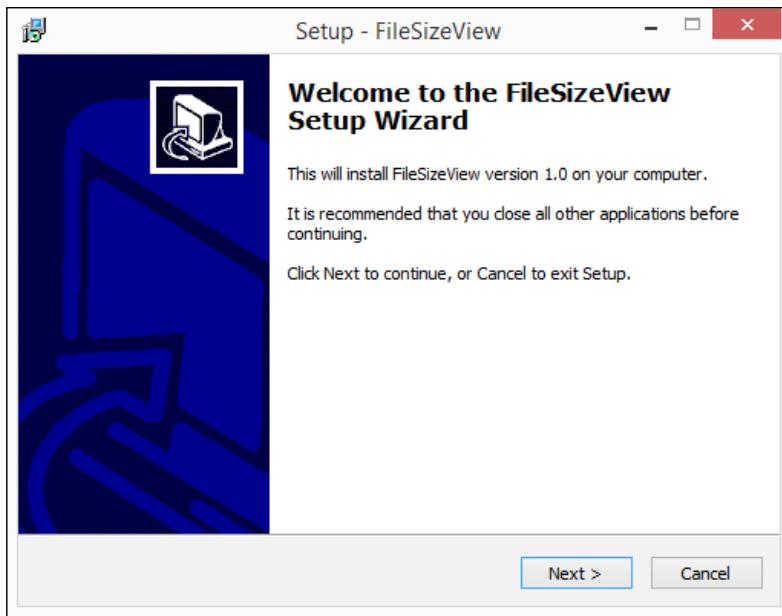
Absolutely; in this instance, better comes in the form of creating a setup installer so that we only need to distribute a single file. This is much easier to work with! It has the added bonus of compressing the files further; in our example, by using the open source Inno Setup package, the results drop from approximately 80 MB to around 30 MB. Let's take a look at what's required to produce a setup file for the Windows platform:

1. We first need to download and install Inno Setup. Head over to <http://www.jrsoftware.org/isinfo.php> and then click on **Download Inno Setup**; the `setup.exe` file can be downloaded from the table about halfway down the page.
2. Double-click on the `setup.exe` file and run through the process, accepting all the defaults.
3. In our project folder, we need to create a new folder called `setup`. This will store the source scripts for Inno Setup and the final builds.

4. From the code download, go ahead and extract `filesizeview-1.0.iss` and store it within the `setup` folder.
5. Double-click on the file to launch it and then click on the highlighted icon, shown in the following screenshot, to compile the build file:



6. When completed, Inno Setup will automatically start the newly created installer, as shown here:



We can now follow through the installation process to completion, before using the application in anger. Inno Setup has also taken care of the uninstallation process, by including a `unins000.exe` file that we can use if we need to remove the application from our system.

For those of you using Mac, there will be similar packages available. Try the instructions listed at <http://www.codepool.biz/tech-frontier/mac/make-pkg-installer-on-mac-os-x.html> as a starting point. You can also try using Inno Setup on Linux, using Wine—the instructions are listed at <http://derekstavis.github.io/posts/creating-a-installer-using-inno-setup-on-linux-and-mac-os-x/>, although they are not for beginners!

Taking things further

Phew! We've certainly covered a lot over the last few pages!

However, in the grand scheme of life, we've only scratched the surface. We can do a lot more with our application or even explore it to help improve our skills when using Node-WebKit with jQuery. To get you started, here are a few ideas:

- The application is a perfect base for resizing images or even compressing them; we can do this online, but there are implications, principally around confidentiality and the size of image.
- The upload facility is only partially working. We use the BlueImp file upload plugin, but it's not actually doing anything. How about getting it working within our application?
- How about displaying an icon for the file type or even a small thumbnail if we're uploading an image?
- There's no way to clear the list without restarting the application – it should be easy to fix this...
- We deliberately didn't include any error checking to keep things simple; how about adding in some now?
- I think the interface is a little limiting in one respect: if we upload a file with a really long name, then it is truncated; the truncation is a little messy!
- We haven't added any menu controls. While Node-WebKit is perfect for applications where speed isn't an issue, it will still be good to be able to navigate around, once we've added more functionality. For an example of how to add such a menu, take a look at <http://www.4elements.com/blog/2013/12>.

Hopefully, here you should find a few ideas to inspire you to take things further. Once you've grasped the basics and allow the occasions where we have to use Node-specific tags, the sky is the limit! A fair few people have produced applications of varying complexity and released them online – it's definitely worth doing some research online to see what is available. Here are a few ideas:

- The Irish developer Shane Gavin has created a useful video-based tutorial on using Node-WebKit. This explores some of the techniques you can use when creating Node-WebKit applications, and we've used some of them in our example. The tutorials are available at <http://nodehead.com/category/video-tutorials/mini-series/node-webkit/>.

- I am sure we've all heard of or played games such as *Pong* or *Breakout* in some form or other. We can use the Phaser game library at <http://phaser.io> to produce some of these classic games (and others). Have a look at the example shown at <https://github.com/kandran/pong> that uses Node-WebKit to create *Pong*.
- David Neumann wrote a blog post about how the free educational game *Caterpillar Count* was repackaged to work in Node-WebKit; leaving aside the nature of the game, the post highlights some useful tips and tricks on the transfer process (<http://blog.leapmotion.com/building-applications-for-simultaneous-deployment-on-web-and-native-platforms-or-how-i-learned-to-stop-worrying-and-love-node-webkit/>).
- Interested in experimenting with your webcam using HTML5 and Node-WebKit? Head over to <http://webcamtoy.com> – it should be relatively easy to adapt the standard code to work from Node-WebKit, as it supports `getUserMedia`.
- If we're working with video or webcams, we can always look at taking screenshots. There is a package available for Node-WebKit to help with this (<https://www.npmjs.com/package/node-webkit-screenshot>); it can easily form the basis of a useful little application.
- We talked earlier about using other JavaScript libraries, such as Ember or Angular, that can be easily used with Node-WebKit and jQuery – for two examples, head over to <http://www.sitepoint.com/building-chat-app-node-webkit-firebase-angularjs/> and <http://sammctaggart.com/build-a-markdown-editor-with-node-webkit-and-ember/>.

There is an increasing amount of content available online. There have been some recent name changes to the library (as was mentioned earlier), so if you want to learn more about using Node-WebKit, then it is worth searching for both Node-WebKit and NW.js to ensure full coverage.

Summary

In recent years, the dividing line between online and offline applications has blurred, with many people using mobile devices to access the Internet in place of normal desktop browsers. With the advent of Node-WebKit, this opens up a lot of opportunities to merge those boundaries even further – let's recap what we learned over the last few pages.

We kicked off with what seems to be a typically simple request, where most developers will automatically think of designing a site. However, with the introduction of Node-WebKit, we can explore creating an offline version of our application or site. We explored a little of how the library works as well as discussed the pros and cons of running such an application from the desktop.

We then moved on to prepare our development environment before taking a brief look at installing Node-WebKit and using it to create our first application. We delved into the package.json file, which is key to running our application, before moving on to build our file size viewer application. Next up came a more in-depth look at the code used behind the application; we also covered how we can create the basic skeleton of our application using the Yeoman Node-WebKit generator.

Next up came a look at a quick tip for debugging Node-WebKit apps, before moving on to examine how we can package and deploy our applications either manually or automate them using Grunt. The final stage in our journey covered the deployment of our application. We looked at using Inno Setup to produce a setup.exe file that can be deployed for use. We then rounded out the chapter with a look at a few ideas of how we can take things further when developing with Node-WebKit.

Phew! We've certainly covered a lot, but as they always say, there is no rest for the wicked. In the next chapter, we will be taking a look at one of the most important parts of using jQuery: optimizing and enhancing the performance of our projects.

13

Enhancing Performance in jQuery

In the book so far, we've covered an array of different topics: from customizing jQuery to the use of animation, and even a little on the use of jQuery within Node-WebKit.

However, there is one key topic we have not yet covered. While working with jQuery can be very fulfilling, we must be mindful of optimizing our code where practical, to ensure a positive user experience. Many developers might simply eyeball the code, but this is time-consuming. In this chapter, we will look at ways of optimizing your jQuery code, introduce the use of tools that can supplement existing workflow, and help give real feedback on your changes. We will cover a number of topics in this chapter, which will include:

- Understanding why performance is important
- Monitoring performance when adding elements
- Monitoring the speed of jQuery
- Automating performance monitoring
- Using Node to lint our code automatically
- Implementing best practices for enhancing performance
- Considering the case of using jQuery

Ready to get started?



Throughout this chapter we will concentrate on using jQuery – you will find that many of the tips given can also be applied to pure JavaScript using it more in your code (as we will discuss later in the chapter).

Understanding why performance is critical

Picture the scene if you will – your team has created a killer web-based application using the latest techniques, which does everything under the sun, and you're ready to sit back and enjoy the laurels of your success. Except for one small but rather critical thing...

No one is buying. Not one copy of your application is being sold – the reason why? Simple – it's really slow and hasn't been properly optimized. No amount of selling will get over the fact that in this age of mobile devices, a slow application will turn off the users.

Should we be concerned with the performance of our application? Absolutely! There are good reasons for being critical of our application's performance; let's take a look at a few:

- The advent of mobile devices with the associated costs of surfing means that our content must be optimized to ensure the site displays quickly, before the connection times out
- It's all too easy to focus on development instead of fixing cross-browser issues – each quirk in itself may not be much, but the cumulative effect will soon add up
- Once you start writing considered code, then it will soon become second nature

Of course, it has to be said that there is a risk of **premature optimization**, where we spend lots of time optimizing code for little gain, and may even cause ourselves problems later if we remove code that is subsequently needed!

Okay – so assuming there is scope to optimize our code, what should we do? Well, there are a few tricks we can use; while we may have the desire to optimize our code ad nauseam, it is not always worth the effort. The smarter approach is to always consider the bigger picture, to make sure that the benefits of optimizing scripts are not lost through badly written style sheets or large pictures, for example!

Let's take a moment to consider some of the options available to us – they include:

- Building custom versions of jQuery
- Minifying our scripts
- Fine-tuning the use of selectors

- Being prudent with event bubbling
- Continuous use of appropriate tools to lint our code
- Minimizing manipulation of the DOM

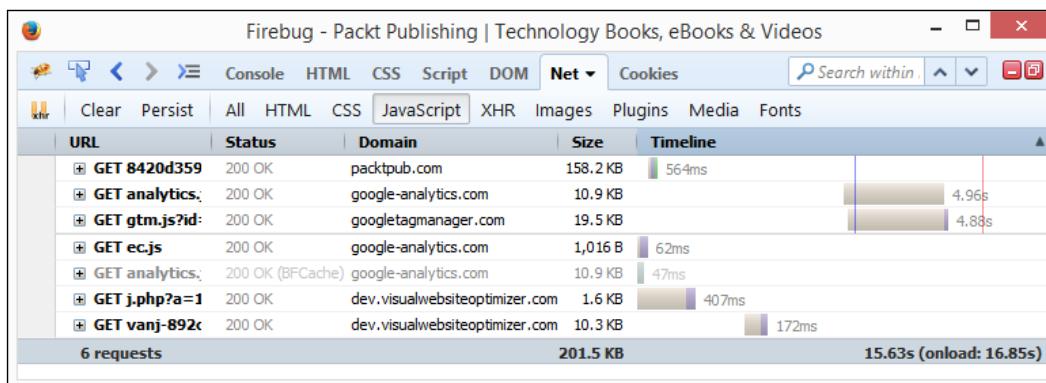
These are some of the options available to us. Our first stop though is to benchmark our code, to see how it performs prior to making any changes. The first step in this is to run a performance check on our scripts. Let's take a moment to see what is involved, and how this works in action.

Monitoring the speed of jQuery using Firebug

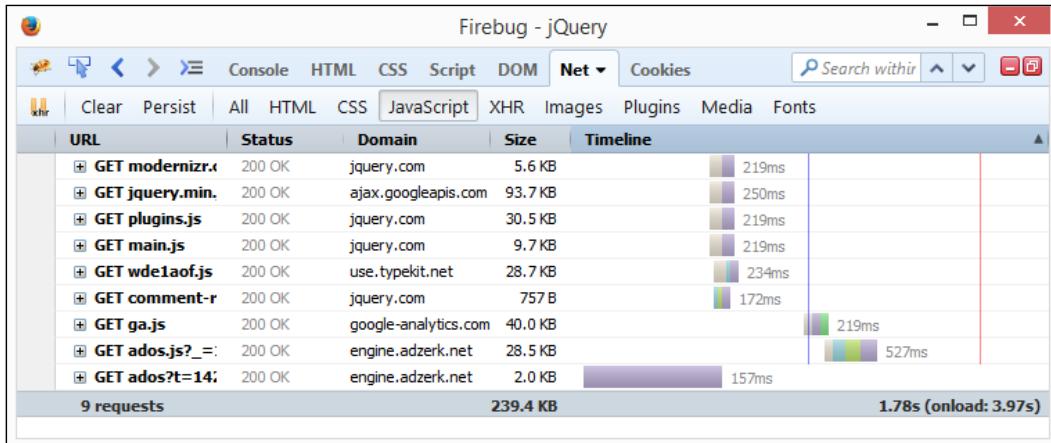
We can wax lyrical about how critical performance is, but nothing beats seeing it in action and working out how we can improve our code to gain that extra edge. Manually working out where to make the changes is time-consuming and inefficient. Instead, we can avail ourselves of a number of tools to help get a clearer indication of where the issues lie in our code.

There are dozens of tools available to help with benchmarking performance of our pages, which include interactions with jQuery or jQuery-based scripts and plugins. Over the next few pages, we're going to look at a selection of different methods. Let's start with a simple visual check, using Firebug, from <http://www.getfirebug.com>. Once installed, click on **Net | JavaScript**, then load your page to get statistics on each plugin or script that is loaded on the page.

In the following image, we can see the results from the Packt Publishing website:



In comparison, following is the image showing the results from <http://www.jquery.com>:



Before loading the page, clear your cache to avoid skewing the results.



Viewing the statistics returned from Firebug gives us a good start, but to get a better indication as to where the bottlenecks are, we need to profile our code. Thankfully, it's a cinch to do with console. Let's take a look at how we can use console to optimize code, using a copy of the tooltipv2.html demo we created in *Chapter 11, Authoring Advanced Plugins*. For the purpose of this little demo, we will run it from a local web server, such as WAMP:

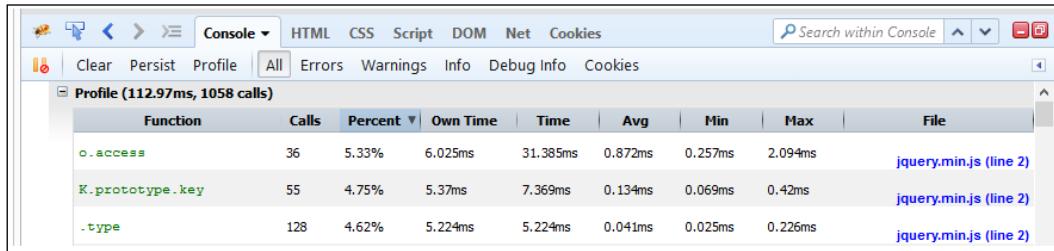
1. From the code download, extract a copy of the tooltip demo folder and store it in the www folder of WAMP.
2. In tooltipv2.js, alter the first few lines as shown next – this adds in the call to profile our code:

```
$ (document) .ready(function() {  
    console.profile();  
    $ ('#img-list li a.tooltips') .quicktip({
```

3. We need to tell the browser when to stop profiling, so go ahead and alter the code as shown next:

```
)  
    console.profileEnd();  
});
```

4. In the browser, load `tooltipv2.html`, and then open Firebug. If all is well, we should see something akin to the following screenshot, where we see the first few lines of the profile report:



Profiling our site using a tool such as Firebug can be very revealing. To give a flavor of how, imagine if we had added more selectors; some of the figures shown would have been much higher then.

[ If you want to focus just on time taken, an alternative to using `console.profile()` is to use `console.time()` and `console.timeEnd()` instead.]

There are many more tools available for profiling our sites. Not all are specific to jQuery, but they can still be used to gain insight into how our scripts are performing. Following are a few examples you can try, in addition to the classic sites such as JSPerf.com (<http://www.jsperf.com>):

- JSLitmus, from <http://code.google.com/p/jslitmus/>
- BenchmarkJS, available at <http://benchmarkjs.com/>, or from the NPM site at <https://www.npmjs.com/package/benchmark> – an example of how to use it is available at <https://gist.github.com/brianjlandau/245674>
- Online services such as SpeedCurve (<http://www.speedcurve.com>), or Calibreapp (<https://calibreapp.com/>)
- FireQuery Reloaded, from <https://github.com/firebug/firequery/wiki> is coming; note that this is still in beta at the time of writing
- DeviceTiming, from <https://github.com/etsy/DeviceTiming>

There are definitely plenty of options available – not all will suit everyone's needs; the key though is to understand what you are testing, and learn how to interpret it.

Dave Methin, part of the core team for jQuery, wrote a brilliant article that outlines the dangers of blindly trying to optimize code, without properly interpreting the results from using something such as JSPerf. The developer Fionn Kelleher puts it perfectly when he states that your code should be a work of art – there is no need to optimize everything for the sake of doing so; it is far more important that code should be readable and work well.

Okay – time to move on. We've covered the basics of monitoring, but at the expense of requiring manual effort. A much better option is to automate it. We can use a number of tools to do this with our old friend Grunt, so let's dig in and see what is involved in automating our monitoring.

Automating performance monitoring

Hands up – as a developer, how many of you have used YSlow? Good – a fair few; have you thought about automating those checks though?

That's right! We can always perform manual checks to get a feel for where performance bottlenecks are showing; however, the smarter way is to automate those checks using our good friend, Grunt. A module, created by the developer Andy Shora, is available for this purpose; we can get the source code for it from <https://github.com/andyshora/grunt-yslow>. Let's take a moment to get it up and running, to see how it works:

1. Let's kick off by creating a project folder for our files. For the purpose of this exercise, I will assume it is called chapter13 (yes, I know – highly original); change the name if yours is different.
2. For this exercise, we need to use NodeJS. I will assume you already have it installed from previous exercises; if not, then head over to <http://www.nodejs.org> to download and install the version appropriate for your platform.
3. Next, add the following to a blank file, saving it as `gruntfile.js` within our project folder – you will notice that our test will be for jQuery's website (as highlighted):

```
'use strict';

module.exports = function (grunt) {
  grunt.initConfig({
    yslow: {
      pages: {
        files: [
          {
            src: 'http://www.jquery.com',
          }],
    },
  });
}
```

```
        options: {
          thresholds: {
            weight: 500,
            speed: 5000,
            score: 90,
            requests: 15
          }
        }
      }
    );
}

grunt.loadNpmTasks('grunt-yslow');
grunt.registerTask('default', ['yslow']);
};
```

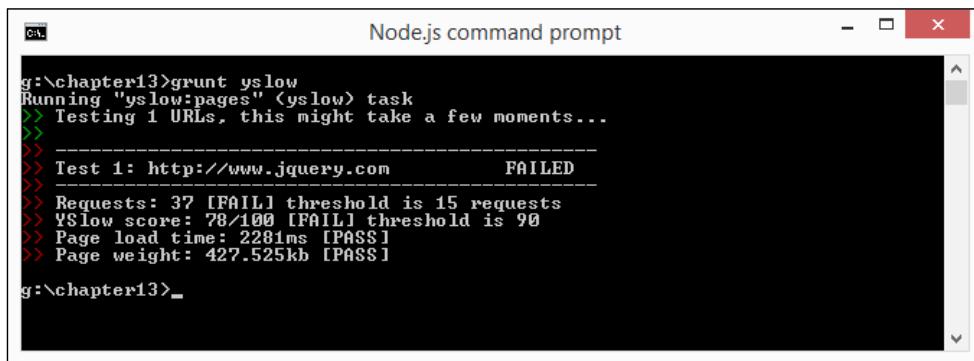
4. In a NodeJS command prompt window, enter the following command to install the `grunt-yslow` package:

```
npm install -g grunt-yslow
```

5. Node will run through the installation. When completed, enter the following command at the command prompt to perform the test:

```
grunt yslow
```

6. If all is well, Node will display something akin to the following screenshot, where it shows a fail:



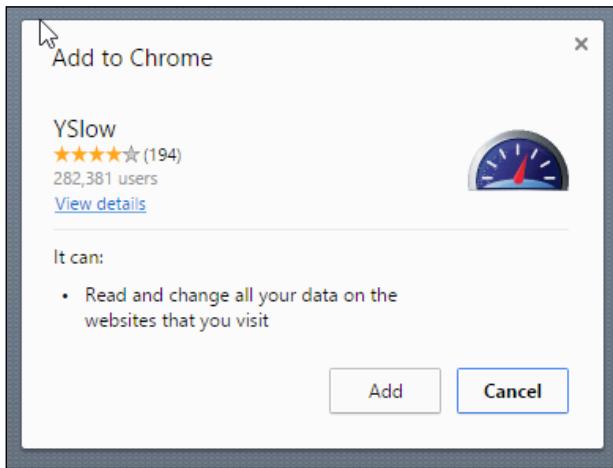
```
g:\chapter13>grunt yslow
Running "yslow:pages" (yslow) task
>> Testing 1 URLs, this might take a few moments...
>>
>> -----
>> Test 1: http://www.jquery.com           FAILED
>>
>> Requests: 37 [FAIL] threshold is 15 requests
>> YSlow score: 78/100 [FAIL] threshold is 90
>> Page load time: 2281ms [PASS]
>> Page weight: 427.525kb [PASS]

g:\chapter13>
```

The results shown in the command prompt window are a little basic. To get a better feel for where the issues are, we can install the YSlow plugin. Let's do that now:

[ At the time of writing, there were ongoing issues with running YSlow in Firefox; please use Chrome to view the results instead. If you are a Mac user, then you can try the YSlow plugin from <http://yslow.org/safari/>.]

1. Browse to <http://www.yslow.org>, then click **Chrome** under **Availability**, and then **Add** to add the plugin to Chrome:



2. Once installed, we can run the report within YSlow. If we do it for the main jQuery site, then we will end up with results similar to those seen in the following screenshot:

Grade C Overall performance score 74 Ruleset applied: YSlow(V2) URL: <http://jquery.com/>

ALL (23) FILTER BY: [CONTENT \(6\)](#) | [COOKIE \(2\)](#) | [CSS \(6\)](#) | [IMAGES \(2\)](#) | [JAVASCRIPT \(4\)](#) | [SERVER \(6\)](#)

F Make fewer HTTP requests

- E Use a Content Delivery Network (CDN)**
- A Avoid empty src or href**
- F Add Expires headers**
- F Compress components with gzip**
- A Put CSS at top**
- C Put JavaScript at bottom**
- A Avoid CSS expressions**
- n/a Make JavaScript and CSS external**

Grade F on Make fewer HTTP requests

This page has 8 external Javascript scripts. Try combining them into one.
This page has 3 external stylesheets. Try combining them into one.
This page has 17 external background images. Try combining them with CSS sprites.

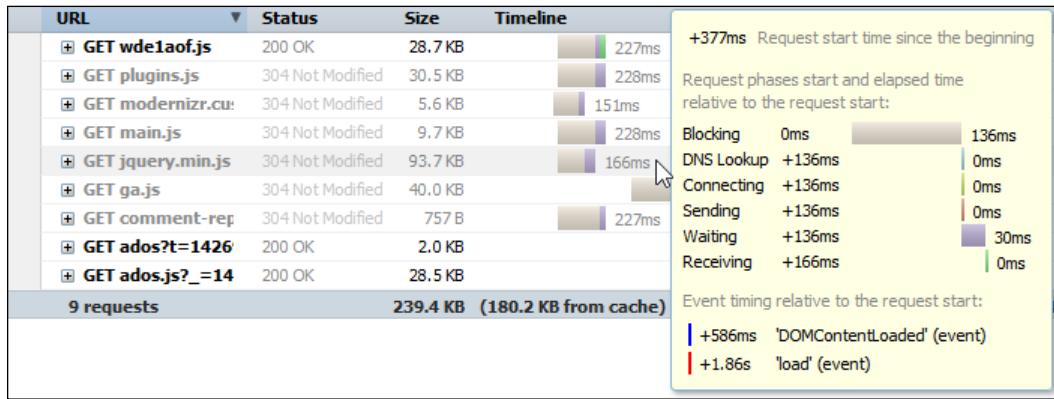
Decreasing the number of components on a page reduces the number of HTTP requests required to render the page, resulting in faster page loads. Some ways to reduce the number of components include: combine files, combine multiple scripts into one script, combine multiple CSS files into one style sheet, and use CSS Sprites and image maps.

[»Read More](#)

Copyright © 2015 Yahoo! Inc. All rights reserved.

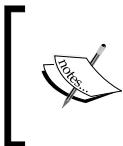
If we take a look through the various grades given, we can clearly see that there is room for improvement. Focusing on the scripts, a check will show that at least five scripts should be moved to the bottom of the page, as the browser cannot start any other downloads until these have been completed.

3. To see what impact this would have, take a look at the same page within Firebug. Click **Net | JavaScript**, then refresh the page to view all the scripts being called from the page. Hover over the jQuery link – this is proof that the bigger the file, the longer it takes to load:



In the previous screenshot, we can clearly see a number of scripts, all of which show long times. In this instance, minifying those scripts that are not already compressed will improve these times.

We can always spend time trying to optimize jQuery, but this should be taken in the context of the bigger picture; we will clearly lose any benefit of optimizing jQuery, if we're still loading large scripts elsewhere in our code.



It's worth noting that the threshold has been set higher than normal within `gruntfile.js`. In this age of mobile devices, it is important to ensure that the page content can be downloaded quickly; in both examples, we will see that there is definitely room for improvement!

Let's take a look at a second example, to see how this compares. In this case, we will use the Packt Publishing website, at `http://www.packtpub.com`:

1. Let's go back to the `gruntfile.js` file that we created at the beginning of this section. We need to modify the following line:

```
files: [{  
    src: 'http://www.packtpub.com',  
},  
options: {
```

2. Save the file, then switch to NodeJS command prompt and enter the following command:

```
grunt yslow
```

3. If all is well, Node will display the results of our assessment of `http://www.packtpub.com`, where we see another failure, as shown in the following screenshot:

```
G:\chapter13>grunt yslow  
Running "yslow:pages" <yslow> task  
» Testing 1 URLs, this might take a few moments...  
»-----  
» Test 1: http://www.packtpub.com      FAILED  
»-----  
» Requests: 45 [FAIL] threshold is 15 requests  
» YSlow score: 81/100 [FAIL] threshold is 90  
» Page load time: 1960ms [PASS]  
» Page weight: 1337.286kb [FAIL] threshold is 500kb  
G:\chapter13>
```

If we take a look using YSlow, as we did before, then we can see a number of suggestions made, which will improve the performance. The key one for us is to condense six scripts into a smaller number of files (and minify them). Refer to the following screenshot:

Grade B Overall performance score 80 Ruleset applied: YSlow(V2) URL: <https://www.packtpub.com/>

ALL (23) FILTER BY: [CONTENT \(6\)](#) | [COOKIE \(2\)](#) | [CSS \(6\)](#) | [IMAGES \(2\)](#) | [JAVASCRIPT \(4\)](#) | [SERVER \(6\)](#)

[Tweet](#) [Share](#)

E Make fewer HTTP requests

F Use a Content Delivery Network (CDN)

A Avoid empty src or href

F Add Expires headers

B Compress components with gzip

A Put CSS at top

A Put JavaScript at bottom

Grade E on Make fewer HTTP requests

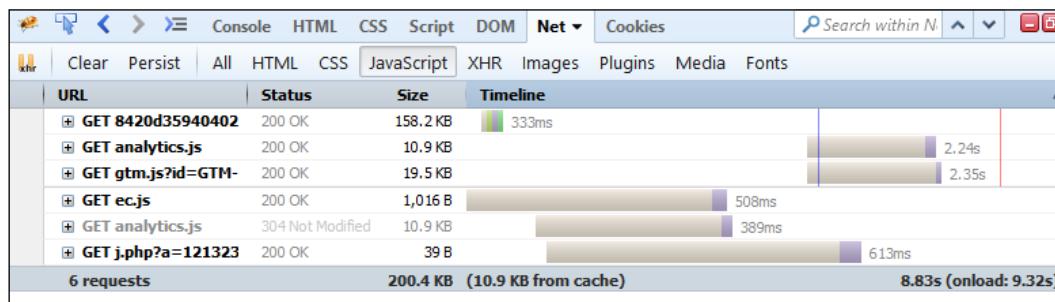
This page has 6 external Javascript scripts. Try combining them into one.

This page has 3 external stylesheets. Try combining them into one.

This page has 15 external background images. Try combining them with CSS sprites.

Decreasing the number of components on a page reduces the number of HTTP requests required to render the page, resulting in faster page loads. Some ways to reduce the number of components include: combine files, combine multiple scripts into one script, combine multiple CSS files into one style sheet, and use CSS Sprites and image maps.

In the previous screenshot, we see similar issues noted by YSlow, although the numbers are not quite as high as on the jQuery website. The real issues show up when we check the timings for loading the scripts called by the main page:



Although we are making fewer requests, which is good, only one of the scripts is minified. This will cancel out the benefits of minimization. We can go some way in rectifying this by minifying the code. We will take a look at how this can be automated later in this chapter, in *Minifying code using NodeJS*.

Gaining insight using Google PageSpeed

So far, we've seen how to monitor pages, but at a very technical level. Our checks have concentrated on the sizes and return times of scripts being called from our page.

A better option is to run a test such as Google PageSpeed, using the Grunt package available from <https://github.com/jrcryer/grunt-pagespeed>; we can see the results in this screenshot:

```
c:\wamp\www\chapter13>grunt pagespeed
Running "pagespeed:prod" <pagespeed> task

URL:      https://www.packtpub.com/books/content/blogs
Score:    83
Strategy: desktop

Number Resources: 34
Number Hosts: 8
Total Request: 4.79 kB
Number Static Resources: 19
HTML Response: 52.64 kB
Text Response: 316.88 kB
CSS Response: 175.33 kB
Image Response: 352.3 kB
JavaScript Response: 690.75 kB
Number JS Resources: 5
Number CSS Resources: 3

Avoid Landing Page Redirects: 0
Enable Gzip Compression: 0.2
Leverage Browser Caching: 1.5
Main Resource Server Response Time: 0
Minify CSS: 0
Minify HTML: 0.11
Minify Java Script: 0
Minimize Render Blocking Resources: 12
Optimize Images: 2.46
Prioritize Visible Content: 2

Done, without errors.
c:\wamp\www\chapter13>
```

It doesn't look at specific scripts or elements on the page, but gives what I would consider to be a more realistic view of how well our page is performing.



This demo requires the use of Node and Grunt, so make sure you have both installed before continuing.

Let's now see it working in action, against the Packt Publishing website:

1. We'll start by firing up a NodeJS command prompt, and then changing to our project folder area.

2. Enter the following to install the `grunt-pagespeed` package:

```
npm install grunt-pagespeed --save-dev
```

3. In a new file, add the following, saving it as `gruntfile.js` in the same folder – there is a copy of this file in the code download; extract and rename `gruntfile-pagespeed.js` to `gruntfile.js`:

`Gruntfile.js:`

```
'use strict';
```

```
module.exports = function (grunt) {
  grunt.initConfig({
    pagespeed: {
      options: {
        nokey: true,
        url: "https://www.packtpub.com"
      },
      prod: {
        options: {
          url:
"https://www.packtpub.com/books/content/blogs",
          locale: "en_GB",
          strategy: "desktop",
          threshold: 80
        }
      }
    }
  });

  grunt.loadNpmTasks('grunt-pagespeed');
  grunt.registerTask('default', 'pagespeed');
};
```

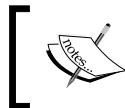
4. At the NodeJS command prompt, enter the following command to generate the report:

```
grunt-pagespeed
```

5. If all is well, we should see a report appear, similar to that shown at the start of our exercise.

The grunt-pagespeed plugin is just one example of several that can be run using Grunt. There are other benchmarking tasks available that we can integrate to continuously monitor our sites. These include the following:

- `grunt-topcoat-telemetry`: Get smoothness, load time, and other stats from Telemetry as part of CI. This could help you set up a performance benchmarking dashboard similar to the one used by Topcoat (<http://bench.topcoat.io>).
- `grunt-wpt`: The grunt plugin for measuring WebPageTest scores.
- `grunt-phantomas`: Response times for requests, responses, time to first image/CSS/JS, on DOM Ready and more.



If you prefer to use Gulp, then the previous Grunt plugins can be run using `gulp-grunt`, available from <https://npmjs.org/package/gulp-grunt>.



Now that we know our baseline, it's time to explore how we can optimize our code; most developers will either eyeball the code manually, or potentially use a site such as www.jshint.com (or even jslint.com). There's nothing wrong in this approach. However, it's not the best approach to take, as it is an inefficient use of our time, which risks missing an opportunity to improve our code.

The smarter way to lint code is to automate the process – while it may not alert you to any earth-shattering changes that need to be made, it will at least ensure that our code doesn't fail optimization due to errors. It will, of course, also give us a solid basis upon which we can make further optimizations. We will cover more of this later in the chapter.

Time for a demo, I think! Let's take a moment to go through setting up the automatic check using NodeJS.

Linting jQuery code automatically

Linting code, or checking it for errors, is an essential part of jQuery development. Not only does it help get rid of the errors, it also helps identify the code that isn't being used within the script.

Don't forget – optimizing isn't just about adjusting selectors or even replacing jQuery code with CSS equivalent (as we saw in *Chapter 6, Animating with jQuery*). We need to first ensure that we have a solid base to work from – we can always do this manually, but the smarter option is to automate the process using a task runner such as Grunt.

Let's take a moment to see how this works in action – note that this assumes you still have NodeJS installed from previous exercises. This time around, we will use it to install the `grunt-contrib-jshint` package, available from <https://github.com/gruntjs/grunt-contrib-jshint>:

1. Setting up the automatic check is very easy. To start, we need to download and install `grunt-contrib-jshint`. Open up a NodeJS command prompt, and enter the following from within the project folder area:

```
npm install grunt-contrib-watch
```

2. Once the installation has completed, go ahead and add the following to a new file, saving it as `gruntfile.js` within the project folder:

```
'use strict';

module.exports = function (grunt) {
    // load jshint plugin
    grunt.loadNpmTasks('grunt-contrib-jshint');

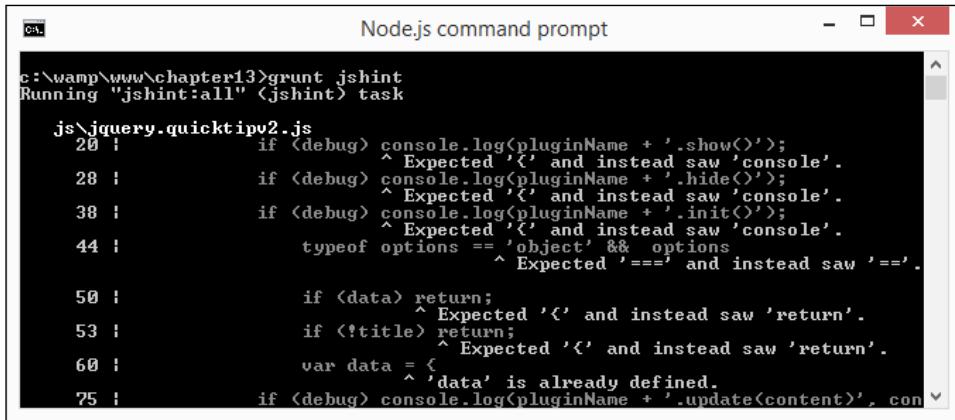
    grunt.initConfig({
        jshint: {
            options: { jshintrc: '.jshintrc' },
            all: [ 'js/script.js' ]
        }
    });

    grunt.loadNpmTasks('grunt-contrib-jshint');
    grunt.registerTask('default', ['jshint']);
};
```

3. From the code download, we need to extract our target JavaScript file. Go ahead and save a copy of `script.js` within a `js` subfolder in our project area.
4. Revert back to the NodeJS command prompt, and then enter the following command to run the `jshint` check over our code:

```
grunt jshint
```

5. If all is well, we should see it pop up three errors that need fixing, as shown in the next screenshot:



The screenshot shows a Windows Command Prompt window titled "Node.js command prompt". The command entered is "grunt jshint". The output shows several JSHint errors found in the file "jquery.quicktipv2.js". The errors are as follows:

```
c:\wamp\www\chapter13>grunt jshint
Running "jshint:all" <jshint> task
js\jquery.quicktipv2.js
 20 :           if (<debug> console.log(pluginName + '.show()');
          ^ Expected '{' and instead saw 'console'.
 28 :           if (<debug> console.log(pluginName + '.hide()');
          ^ Expected '{' and instead saw 'console'.
 38 :           if (<debug> console.log(pluginName + '.init()');
          ^ Expected '{' and instead saw 'console'.
 44 :             typeof options == 'object' && options
          ^ Expected '===' and instead saw '=='.
 50 :               if (<data> return;
          ^ Expected '(' and instead saw 'return'.
 53 :               if (!title) return;
          ^ Expected '(' and instead saw 'return'.
 60 :                 var data =
          ^ 'data' is already defined.
 75 :                   if (<debug> console.log(pluginName + '.update(content)', con
```



The observant of you may spot that this is the code from the quicktip plugin we created back in *Chapter 11, Authoring Advanced Plugins*.

We can take this even further! Instead of manually running the check when code has been updated, we can ask Grunt to do this automatically for us. To make this happen, we need to install the `grunt-contrib-watch` package, and alter the Grunt file accordingly. Let's do that now:

1. Open a copy of `gruntfile.js`, and then add the following code immediately before the closing `) ;` of the `grunt.initConfig` object:

```
},
watch: {
  scripts: {
    files: ['js/script.js'],
    tasks: ['jshint'],
    options: { spawn: false }
  }
}
```

2. At the end of the file, add the following line, to register the additional task:

```
grunt.loadNpmTasks('grunt-contrib-jshint');
```

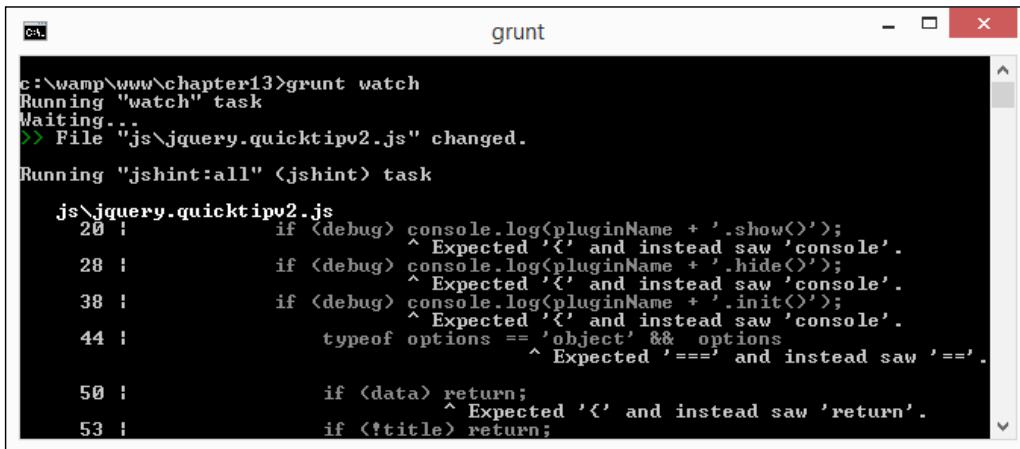
3. We need to alter the `registerTask` call to make Grunt aware of our new task. Go ahead and modify as shown:

```
grunt.registerTask('default', ['watch', 'hint']);
```

4. Switch back to the command prompt window, and then enter the following at the command line:

```
grunt watch
```

5. Switch back to `script.js`, and make a change somewhere in the code. If all is well, Node will kick in and recheck our code.



```
c:\wamp\www\chapter13>grunt watch
Running "watch" task
Waiting...
>> File "js\jquery.quicktipv2.js" changed.

Running "jshint:all" <jshint> task

js\jquery.quicktipv2.js
 20 :           if (<debug> console.log(pluginName + '.show()'));
          ^ Expected '{' and instead saw 'console'.
 28 :           if (<debug> console.log(pluginName + '.hide()'));
          ^ Expected '{' and instead saw 'console'.
 38 :           if (<debug> console.log(pluginName + '.init()'));
          ^ Expected '{' and instead saw 'console'.
 44 :               typeof options == 'object' && options
          ^ Expected '===' and instead saw '=='.
 50 :               if (<data> return;
          ^ Expected '{' and instead saw 'return'.
 53 :                   if (!title) return;
```

Running the code clearly shows that we have some issues we need to fix. At this stage, we would spend time fixing them. As soon as changes are made, Node will kick in and show an updated list of errors (or a pass!).

Assuming our code is fit for purpose, we can really start with optimizing it. An easy win is to minify the code, to help keep file sizes low. We can of course manually compress it, but that is so old-school; time to dig out Node again!

Minifying code using NodeJS

A key part of any developer's workflow should be a process to minify the scripts used in a site. This has the benefit of reducing the size of the downloaded content to a page.

We can of course do this manually, but it's a time consuming process which adds little benefit; a smarter way is to let NodeJS take care of this for us. The beauty of doing this means that we can configure Node to run with a package such as `grunt-contrib-watch`; any changes we make would be minified automatically. There may even be occasions when we decide not to produce a minified file; if we're unsure that the code we are writing is going to work. At times like this, we can instead fire off Grunt from within our text editor, if we're using a package such as Sublime Text.



If you want to implement that level of control within Sublime Text, then take a look at `sublime-grunt`, available from <https://github.com/tvooo/sublime-grunt>.

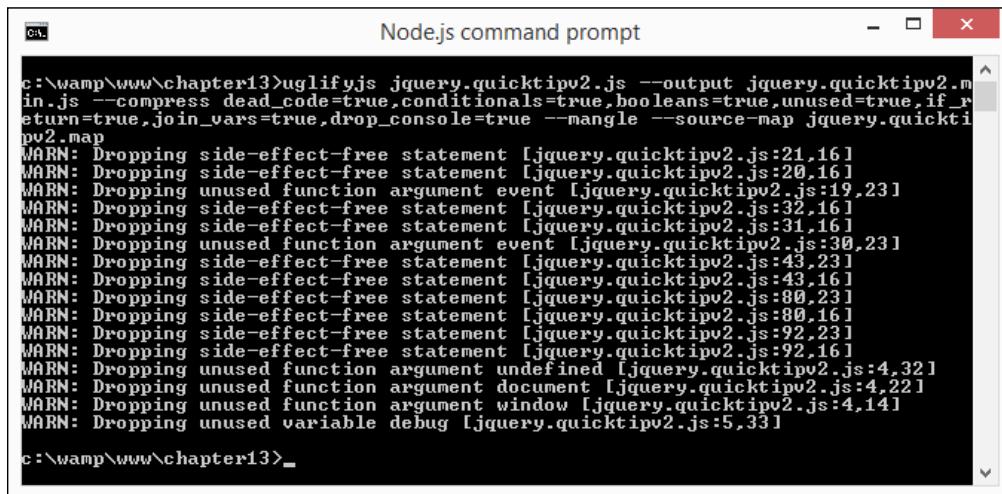
Okay, let's start with setting up our minification process. For this, we'll use the well-known package, `UglifyJS` (from <https://github.com/mishoo/UglifyJS2>), and get Node to automatically check for us:

1. We will be using NodeJS for this demo, so if you haven't already done so, go ahead and download the appropriate version for your platform from <http://www.nodejs.org>, accepting all defaults.
2. For this demo, we need to install two packages. `UglifyJS` provides support for source maps, so we need to install this first. From a NodeJS command prompt, change to the project folder, enter the following command, and then press *Enter*:
`npm install source-map`
3. Next, enter the following command, and press *Enter*:
`npm install uglify-js`

4. When the installation has completed, we can run `UglifyJS`. At the command prompt, enter the following command carefully:

```
uglifyjs js/jquery.quicktipv2.js --output js/jquery.quicktipv2.min.js --compress dead_code=true,conditionals=true,booleans=true,unused=true,if_return=true,join_vars=true,drop_console=true --mangle --source-map js/jquery.quicktipv2.map
```

5. If all is well, Node will run through the process, similar to this next screenshot:



The screenshot shows a Windows command prompt window titled "Node.js command prompt". The command entered was:
`c:\wamp\www\chapter13>uglifyjs jquery.quicktipv2.js --output jquery.quicktipv2.min.js --compress dead_code=true,conditionals=true,booleans=true,unused=true,if_return=true,join_vars=true,drop_console=true --mangle --source-map jquery.quicktipv2.map`

The output displayed in the window is as follows:

```
MARN: Dropping side-effect-free statement [jquery.quicktipv2.js:21,16]
MARN: Dropping side-effect-free statement [jquery.quicktipv2.js:20,16]
MARN: Dropping unused function argument event [jquery.quicktipv2.js:19,23]
MARN: Dropping side-effect-free statement [jquery.quicktipv2.js:32,16]
MARN: Dropping side-effect-free statement [jquery.quicktipv2.js:31,16]
MARN: Dropping unused function argument event [jquery.quicktipv2.js:30,23]
MARN: Dropping side-effect-free statement [jquery.quicktipv2.js:43,23]
MARN: Dropping side-effect-free statement [jquery.quicktipv2.js:43,16]
MARN: Dropping side-effect-free statement [jquery.quicktipv2.js:80,23]
MARN: Dropping side-effect-free statement [jquery.quicktipv2.js:80,16]
MARN: Dropping side-effect-free statement [jquery.quicktipv2.js:92,23]
MARN: Dropping side-effect-free statement [jquery.quicktipv2.js:92,16]
MARN: Dropping unused function argument undefined [jquery.quicktipv2.js:4,32]
MARN: Dropping unused function argument document [jquery.quicktipv2.js:4,22]
MARN: Dropping unused function argument window [jquery.quicktipv2.js:4,14]
MARN: Dropping unused variable debug [jquery.quicktipv2.js:5,33]
```

The command prompt then ends with:
`c:\wamp\www\chapter13>`

6. At the end, we should have three files in our project area, as shown in the following screenshot:

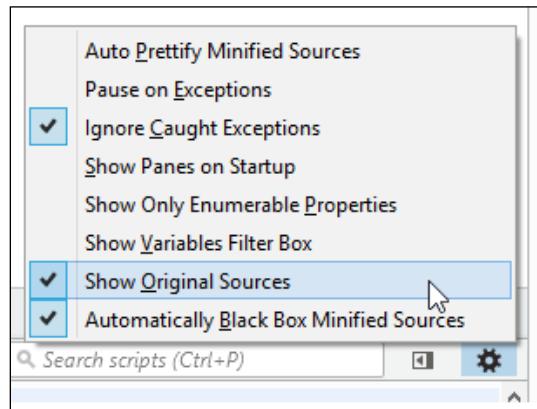
 jquery.quicktipv2.js	22/03/2015 15:15	JavaScript File	5 KB
 jquery.quicktipv2.map	22/03/2015 18:13	MAP File	3 KB
 jquery.quicktipv2.min.js	22/03/2015 18:13	JavaScript File	2 KB

We're now free to use the minified version of our code within a production environment. While in this instance we've not made much of a saving, you can imagine the results if we were to scale these figures up to cover larger scripts!

Exploring some points of note

The process of compressing scripts should become a de facto part of any developer's workflow. NodeJS makes it easy to add, although there are some tips that will help make compressing files easier and more productive:

- The default configuration for UglifyJS will only produce files that show little compression. Getting better results requires careful reading of all the options available, to get an understanding of which one may suit your needs and is likely to produce the best results.
- We've included the source map option within our compression process. We can use this to relate issues appearing to the original source code. Enabling source map support will differ between browsers (for those that support it); in Firefox for example, press *F12* to show the Developer Toolbar, then click on the cog and select **Show Original Sources**:



- It is worth checking to see if minified versions of files used in your project are already available. For example, does your project use plugins where minified versions have already been provided? If so, then all we need to do is concatenate them into one file; minifying them again is likely to cause problems, and break functionality in the file.

Minifying files is not a black art, but is equally not an exact science too. It is difficult to know what improvement you will get in terms of file size, before compressing them. You may get some results that you didn't expect to see. It's worth exploring one such example now.

Working through a real example

While researching material for this book, I tried minifying one of the Drupal files used on the Packt Publishing site as a test. The original weighed in at 590 KB; a compressed version using the same configuration options as in our demo, produced a file that was 492 KB.

What does this tell us? Well, there are a couple of things to note:

- It is important to maintain a realistic sense of expectation. Compressing files is a useful trick we use, but it will not always produce the results we need.
- We've used UglifyJS (version 2). This is really easy to use, but comes with a trade-off in terms of raw compression ability. There will be some instances where it won't suit our requirements, but this shouldn't be seen as a failing. There are dozens of compressors available; we simply will have to choose a different alternative!
- To really get a significant reduction in size, it may be necessary to use gzip to compress the file, and configure the server to decompress on the fly. This will add an overhead to processing the page, which needs to be factored into our optimization work.

Instead, it may be a better alternative to work through each script to determine what is and isn't being used. We can of course do this manually, but hey – you know me by now: why do it yourself when you can put it off to something else to do it for you (to badly misquote a phrase)? Enter Node! Let's take a look at `unusedjs`, which we can use to give us an indication of exactly how much extra code our scripts contain.



We've concentrated on minifying one file, but it is a cinch to change the configuration to minify any file automatically, by using wildcard entries instead.

Working out unused JavaScript

So far, we've seen how we can easily minify code without any effort – but what if minifying isn't enough, and we need to remove redundant code?

Well, we can manually eyeball the code – nothing wrong with that. It's a perfectly acceptable way of working out what we can remove. The key thing though is that it is a manual process, which requires a lot of time and effort – not to mention the frequent attempts to find code that we can remove without breaking something else!

A smarter move is to set Node to work out for us what is being used, and what could be safely dropped. The web performance expert Gaël Métais has created unused JS to help with this. It works with Node, and is available at <https://www.npmjs.com/package/unusedjs>. It's a work in progress, but as long as it is used as a guideline, it can produce a useful basis for us to work out where we can make changes.

Let's take a moment to dig in and see how it works. For this demo, we'll use the Tooltip plugin demo we created in *Chapter 12, Using jQuery with the Node-WebKit Project*.



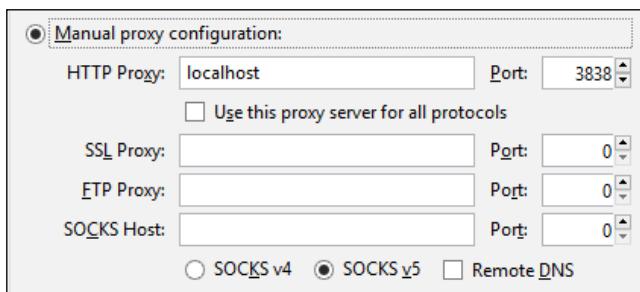
There are a few things that we need to bear in mind when using this functionality:

- At the time of writing, the status of this plugin is still very much alpha – the usual risks around using alpha software apply! It is not perfect; it should be used as a guideline only, and at your own risk. It doesn't work well with really long scripts (such as the jQuery UI library), but will manage around 2,500-3000 lines.
- You will need to clear your browsing history, so don't go and use it in a browser where maintaining history is important.
- The plugin uses Node. If you don't have this installed, then head over to the Node site at <http://www.nodejs.org> to download and install the version appropriate for your platform.

- We also need to use a local web server such as WAMP (for PC – `http://www.wampserver.com/de` or `http://www.wampserver.com/en/`), or MAMP (for Mac – `http://www.mamp.info`) for the demo. Make sure you have something set up and configured for use.

Assuming we have Node and a local web server installed and configured for use, let's start with setting the `unusedjs` script. We will use Firefox for the purpose of running the demo, so adjust accordingly if you prefer to use a different browser. Let's begin:

1. We need to start somewhere. The first step is to install `unusedjs`. Run the following command at the NodeJS prompt:
`npm install unusedjs -g`
2. Start the server by writing the following in your console:
`unusedjs-proxy`
3. Click on the three bar icon and then **Options**, to show the options dialog. Make sure the following entries are set as shown in this next image:



4. Make sure the **No Proxy** field is empty. Then click **OK** to confirm the settings.
5. Next, we need to clear the cache in the browser session. This is critical, as we will likely get skewed results if the cache is not cleared.
6. At this stage, open a copy of `tooltipv2.html` from the code download that accompanies this book, and wait until the page is fully loaded.
7. Press `F12` to display Firefox's console, and enter the following at the prompt:
`_unusedjs.report()`

8. If all is well, we should see something akin to the following screenshot, when viewing the console results:

```
"File 1: " 71.9% of unused code for: http://localhost/chapter13      jquery.js:57:0
/js/jquery.js (over 3214 statements)"
"File 2: " 52.2% of unused code for: http://localhost/chapter13      jquery.js:57:0
/js/jquery.quicktipv2.min.js (over 23 statements)"
"File 3: " 46.2% of unused code for: http://localhost/chapter13      jquery.js:57:0
/js/tooltipv2.js (over 13 statements)"
" Total: 71.7% of unused code (for the moment)"      jquery.js:73:20
```

Try entering `_unusedjs.file(2)` in the console. This function shows a copy of the code, with unused sections highlighted in red, as shown in this screenshot:

```
" 9181. jQuery.noConflict = function( deep ) {
" 9182.   "   "if ( window.$ === jQuery ) {
" 9183.     "   window.$ = _$;
" 9184.   "
" 9185.
" 9186.   "   "if ( deep && window.jQuery === jQuery ) {
" 9187.     "   window.jQuery = _jQuery;
" 9188.   "
" 9189.
" 9190.   "   "return jQuery;
" 9191. };
" 9192.
```

We can now concentrate on the highlighted sections to remove redundant code from our own scripts. How much will of course depend on our own requirements, and whether redundant code will later be used as part of any forthcoming changes to our work.



It goes without saying that we can't simply yank out code from a library such as jQuery. We would need to build a custom version of jQuery – we covered this in detail in *Chapter 1, Installing jQuery*.

Now that we've established our baseline, and worked out if any of our scripts contain unused code, it's time to look at optimizing it. Let's take a look at some of the tips and tricks we can use in our code; as a basis for embedding best practice into our normal development workflow.

Implementing best practices

Imagine the scenario – we've written our code and checked it to ensure that all files are minimized where possible, and that we've not included lots of redundant code. At this point, some might think that we're ready to release the code and put our efforts out for public consumption, right?

Wrong! It would be remiss to release code at this stage, without reviewing our code for both speed and efficiency. Larry Page, cofounder and CEO of Google, put it perfectly, when he stated that :

"As a product manager you should know that speed is product feature number one."

-Larry Page, co-founder and CEO of Google

Speed is absolutely king! We've gone some way towards satisfying Larry's comment, but we can do more. So far, we've looked at minifying our code and producing custom versions of jQuery. We can take this further by assessing the code we've written, to ensure it is being executed efficiently. Each person's requirements will naturally be different, so we would need to use a mix of tricks to ensure efficient execution. Let's take a look at a few:

1. It goes without saying, but we should execute a task against the DOM only when absolutely necessary. Each hit on the DOM can potentially be expensive on resources, making your application slower. For example, consider the following code:

```
<script src="jquery-2.1.3.min.js"></script>
<script type="text/javascript">
    $("document").ready(function() {
        console.log("READY EVENT (B) => " + (new Date() .
        getTime() - performance.timing.navigationStart) + 'ms');
    });
    console.log("END OF HEAD TAG (A) => " + (new Date()
        .getTime() - performance.timing.navigationStart) + 'ms');
</script>
```

2. On an empty <body> tag, the time taken to load the jQuery library and make it available for use, is relatively less; as soon as we add elements to our page, that value will increase. To see the difference, I ran a small demo using this code. In the following image, the results of loading jQuery on empty <body> tags is on the left, while the results of using tooltipv2.html from an earlier demo, on the right:



3. If version 1.11 of jQuery is used, then the effect is even more acute, as a result of the code incorporated to support older browsers. To see the effects for yourself, try running `test loading jquery.html`, and then switch to **Console** within the Developer Toolbar of your browser to see the results of the test. Change the version of jQuery to `1.11` to really see the difference!

To maintain performance, DOM elements should be cached in variables, then append it only after it has been manipulated:

```
// append() is called 100 times
for (var i = 0; i < 100; i++) {
    $("#list").append(i + ", ");
}

// append() is called once
var html = "";
for (var i = 0; i < 100; i++) {
    html += i + ", ";
}
$("#list").append(html);
```



You can see the results in action by running the test on JSPerf, at <http://jsperf.com/append-on-loop/2>.

4. The flip side to this is if we need to modify several properties that relate to a single element. In some respects, creating an object makes it easier to manipulate it, but at the same time undoes all our effort!
5. It is imperative that you check your selectors. jQuery reads them from right to left. Where possible, use IDs as they are faster to access than standard class selectors. Also, make sure that you are not using rules such as `.myClass1 #container`, where an ID follows a class selector. This is inefficient – we lose the benefit of specifying what can only be a single value, by having to constantly iterate through code to ensure we've covered all instances of the class that we have used in our code.

It goes without saying that any selector used should be cached. When referencing selectors that go several levels deep, best practice states that we should be as specific as possible on the left side (that is, `.data`), and less specific on the right:

```
// Unoptimized:  
$( ".data .gonzalez" );  
  
// Optimized:  
$( "div.data td.gonzalez" );
```

6. Above all, avoid using the universal selector in the form of a `*` or type, such as `:radio`, without making your selector reference as explicit as possible – these are both very slow!
7. Although this book is about jQuery, there may be instances where we need to use classic JavaScript methods if performance is such that jQuery is not up to the mark. For instance, a `for` loop will be more efficient than the jQuery `.each()` method, and using the `querySelector` API is better than using a jQuery selector.
8. If you are loading a number of scripts, consider loading them at the end of the page, once all the content has been loaded above the fold (or what is displayed before scrolling down the page). jQuery should always be used to progressively enhance pages, not run a code that will break a page if jQuery is disabled. Perception can play a big part – your page may not be doing a great deal, but still be perceived as slow. Reordering scripts (and content) can help alter this perception.
9. Some developers may still use jQuery's AJAX object to handle asynchronous HTTP requests. Although it works, it is not the cleanest way to manage such requests:

```
$.ajax({  
    url: "/firstAction",  
    success: function() {  
        //whatever  
        secondAction();  
        return false;  
    },  
    error: error()  
});  
  
var secondAction = function() {  
    $.ajax({  
        url: "/secondAction",  
    });  
};
```

```

        success: function() {
            // whatever
        },
        error: error()
    );
}
;

```

A smarter option is to use `jQuery promises()`, where we can defer code into functions that are cleaner to read and easier to debug. It matters not one jot where the code is then stored; `promises()` will allow us to call it at the appropriate point in the code:

```

$.when($.ajax( { url: "/firstAction" } ))

// do second thing
.then(
    // success callback
    function( data, textStatus, jqXHR ) {},
    // fail callback
    function(jqXHR, textStatus, errorThrown) {}
)

// do last thing
.then(function() {});

```

10. If we're calling whole scripts, then it makes sense to explore the use of conditional loaders such as RequireJS (using plain JavaScript), or `grunt-requirejs` (if our preference is to use Node).



It goes without saying that the same principle of lazy loading our code will also apply to elements on the page, such as images; `jquery-lazy` is a perfect example of a module for Node, which will help with this. It's available at <https://www.npmjs.com/package/jquery-lazy>.

11. The previous tip mentioning the use of `promises()` illustrated a perfect example of where we can still make improvements. Some developers extol the virtues of chaining code, which can appear to shorten the code. However, it makes it harder to read and therefore debug; the resulting code spaghetti will lead to mistakes and time wasting, ultimately requiring a partial or complete code refactor. The example from the previous tip also highlights the need to ensure that a good naming convention is used, as we can't be specific with our callback function names, when chaining commands.

12. This next tip may seem a little contradictory, given we are talking about jQuery throughout this book, but use less JavaScript – anything that can be offloaded onto HTML or CSS will have a positive impact on our performance. Although it is fun to use jQuery, it is based on JavaScript which is the most brittle layer of the web stack, and will impact performance. A classic example of this is creating animations. Take a look at <https://css-tricks.com/myth-busting-css-animations-vs-javascript/> to understand why it is foolish to use jQuery (or JavaScript) to power our animations unless necessary.
13. Consider cutting the mustard, or dropping functionality for less capable browsers. This will make for a much better experience when using jQuery-based sites on less capable or mobile browsers. On some sites that have lots of polyfills running to support functionality such as CSS3 styling, the impact of dropping these polyfills could be substantial!
14. To get a feel for the difference in times for loading and parsing jQuery, check out the tests performed by developer Tim Kadlec, at <http://timkadlec.com/2014/09/js-parse-and-execution-time/>.

There are many more tips and tricks we can use in our code. For more sources of inspiration, take a look at the following links as a starting point:

- <http://www.slideshare.net/MatthewLancaster/automated-perf-optimization-jquery-conference>: Presented at the jQuery Conference by developer Matthew Lancaster in 2014, this covers some useful tips; he particularly makes a point of emphasizing that we can make some serious gains with little effort, although we should always be wary of over-optimizing our code!
- <http://crowdfavorite.com/blog/2014/07/javascript-profiling-and-optimization/>: This article goes through the process used by the authors to help optimize performance; this gives a flavor of what is involved.
- <http://chrisbailey.blogs.ilrt.org/2013/08/30/improving-jquery-performance-on-element-heavy-pages/>: This article is a little older, but still contains some useful pointers for optimizing our code.
- <http://joeydehnert.com/2014/04/06/9-development-practices-that-helped-me-write-more-manageable-and-efficient-javascript-and-jquery/>: This contains some very useful tips on optimizing jQuery, with some similar to the ones we've covered in this chapter.

The key point through all of this is that performance optimization should never be considered a one-off exercise; we must consider it an ongoing process during the life of the code. To help with this, we can design a strategy to stay on top of optimization. Let's use some of these tips as a basis for what we need to consider as part of this strategy.

Designing a strategy for performance

So far, we've concentrated on tips and tricks we can use to improve performance. Taking a reactive approach will work, but requires extra time to be spent when we can instead build in these tips and tricks at the time of creating the code.

With this in mind, having a strategy to help encourage such a mindset will help. Let's take a look at a few key points that can form the basis for such a strategy:

- Always use the latest version of jQuery – you benefit from improvements in code, speed, and bug fixes for known issues.
- Combine and minify scripts where possible, to reduce bandwidth usage.
- Use native functions instead of jQuery equivalents – a perfect example is to use `for()` instead of `.each()`.
- Use IDs instead of classes – IDs can only be assigned once, whereas jQuery will hit the DOM multiple times looking for each class, even if only one instance exists.
- Give selectors a context. Refer to the following code simply specifying a single class:

```
$('.class').css ('color' '#123456');
```

Instead, a smarter way is to use contextualized selectors in the form of `$(expression, context)`, thus yielding:

```
$('.class', '#class-container').css ('color', '#123456');
```

The second option runs much faster, as it only has to traverse the `#class-container` element and not the entire DOM.

- Cache values wherever possible, to avoid manipulating the DOM directly.
- Use `join()` instead of `concat()` to join longer strings.
- Always add `return false`, or use `e.preventDefault()` on click events on links where `#` is used as the source link – not adding it will jump you to the top of the page, which is irritating on a long page.
- Set yourself a budget in terms of page weight, requests, and render time – see <http://timkadlec.com/2014/11/performance-budget-metrics/>. It gives purpose to optimizing, and encourages a longer term performance monitoring spirit.
- Use a performance monitoring service such as SpeedCurve to help monitor your sites, and alert you to any issues when they appear.

- Put performance on display in your office – this helps encourage a team spirit. If someone in the team comes up with a change that has a positive impact on performance, then credit them and make the rest of the team aware; it will help encourage a healthy sense of competition amongst the team.
- If however a change breaks the performance, then don't punish the culprit; this will discourage them from taking part. Instead, try to foster a culture of owning a problem, then learning how to prevent it from happening again. How about running tests such as PhantomJS to help check and minimize the risk of issues appearing?
- Automate everything. There are services that will compress images or shrink scripts, but there is something to be said for investing time in developing similar processes in-house that will save you time and money. The key here is that there is no point in manually performing tasks such as optimizing images or minifying scripts; it's up to you to work out what best suits your needs.
- A key consideration is if you decide to use Grunt or Gulp – will either provide additional functionality that is useful, or are they simply an overhead that can be reduced or eliminated with careful use of NPM? The developer Keith Cirkel put together a valid argument for just using NPM at <http://blog.keithcirkel.co.uk/why-we-should-stop-using-grunt/>; it's a thought-provoking read!
- Spend time influencing your colleagues and those higher up the chain – often they may not be aware of the pain you might be experiencing on an issue, but may actually be in a position to help you in your fight!
- Spend time learning. All too often we spend too much time on client work, and don't build in enough time for self-development; put aside some time to rectify this. If it means having to alter prices to cover lost earnings as a result of time not spent on client work, then this is something that needs to be considered. It's all about setting a work/play/learning balance, which will pay off in the long term.

There is plenty of food for thought – not every trick will apply. There will be instances where a blend of one or more will produce the results you need. It is worth spending time on this though, as it will pay off in spades in the longer term, and hopefully will become embedded in existing work culture within your team.

Let's move on. We're almost at the end of the chapter, but before we finish and take a look at testing for jQuery, I want to ask a simple question: do we really need to use jQuery at all, and if so, why should we?

Staying with the use of jQuery

At this point, you would be forgiven for thinking that I've completely lost the plot, particularly when we've just been examining ways of optimizing it, only to suggest that we completely remove its use from our code. Why, I hear you ask, would I even consider dropping jQuery?

Well, there are several good reasons for this. Anyone can write jQuery code, but the smart developer should always consider if they should use jQuery to solve a problem:

- jQuery is an abstraction library. It needs JavaScript, and was built at a time when developing for browsers of the day could be a real challenge. The need to abstract away browser inconsistencies is becoming less and less. It's important to remember that we should use jQuery to progressively enhance plain JavaScript; jQuery was first and foremost designed to make writing JavaScript easier, and is not a language in its own right.
- Browsers are closer than they've ever been, in terms of offering functionality. With Firefox having ditched most vendor prefixes, there is little need for libraries to smooth out inconsistencies. If something works in IE10 or the latest version of Firefox, then it is likely the same will apply for Google Chrome or Opera. Granted, there will be some differences, but this is really only for some of the more esoteric CSS3 styles that have yet to make it into mainstream use. So - if browsers are this close, why use jQuery?
- Using plain JavaScript will always be faster than jQuery, no matter how much we try - it has the added bonus that JavaScript code will produce a smaller file than equivalent JavaScript code (not including the library itself). If we're only using a small amount of JavaScript code, then why reference a whole library? Absolutely, we can always try to build a custom version of jQuery, as we saw back in *Chapter 1, Installing jQuery* - but we're still going to be pulling in more than we need, no matter how much we try to trim unnecessary code from the library! We can of course use gzip to compress jQuery code even further, but it will still be more than plain JavaScript.
- jQuery is all too easy to write - it has a huge community, and the learning curve is low. This creates the perfect conditions for writing lots of low-quality code, where we only use a small subset of the features available in jQuery. It will be much better in the long run to learn how to use plain JavaScript effectively, and then use jQuery to provide the metaphorical icing on the cake.

The key point here though is that we shouldn't completely drop jQuery - the time has come though to really consider if we need to use it.

Granted – if we're using a significant amount of functionality that would otherwise be awkward at best, or positively awful in plain JavaScript, then it will be necessary to use jQuery. However, I'll leave you with a challenge, using taking photos as an analogy. Compose your photo as normal. Then stop, close your eyes for ten seconds, and take a couple of deep breaths. Now ask yourself if you are ready to still take the same photo. Chances are you will change your mind. The same thing applies to using jQuery. If you stopped and really considered your code, how many of you would still decide to continue using it, I wonder?

Personally, I think jQuery will still have a part to play, but we're at a point where we should not simply use it blindly or out of habit, but make a conscious decision about when and where we use it in place of plain JavaScript.

To get a feel for how to switch from jQuery to JavaScript for the simple requirements, take a look at the article by Todd Motto at <http://toddmotto.com/is-it-time-to-drop-jquery-essentials-to-learning-javascript-from-a-jquery-background/>.

Summary

Maintaining performant sites is a key part of development. There is more to it than just optimizing code, so let's take a moment to review what we've learnt in this chapter.

We kicked off with a look at the reasons for understanding why performance is critical, before going through various ways to monitor performance, from the simple eyeballing of statistics in Firebug to automating our checks using Grunt.

We then moved on to understand how we can lint our code automatically, as one of the many ways of optimizing our code, before minifying it for production use. We then dived off to take a look at how we can work out if our code contains any unused code, which can be safely removed as part of streamlining our code.

We then rounded up the chapter with a look at implementing best practices. The focus here was less on providing specific examples, and more on sharing some tips and tricks that can be applied to any site. We then used this as a basis for designing a strategy to help maintain performant sites.

We're almost through our journey in mastering jQuery, but before we finish, we need to take a quick look at testing our code. Developers are likely to use QUnit, given it is part of the same jQuery family of projects; we'll take a look at how we can take things further in the next chapter.

14

Testing jQuery

To test or not to test, that's the question...

To paraphrase that world-famous detective, the answer to this question should be elementary!

If you've spent any time with jQuery, you will no doubt be aware of its unit: the need to test code, and that the most popular way is to use its testing library, QUnit. Throughout this chapter, we'll recap how to use it and then look at some of the best practices we should use as well as explore how we can really cut down our workflow effort, by automating the tests we perform on our code.

In this chapter, we'll cover the following topics:

- Revisiting QUnit
- Automated testing using NodeJS and RequireJS
- Best practices when using QUnit

Are you ready to get stuck in? Let's get started...

Revisiting QUnit

Testing any code is vital to the successful construction of any online application or site; after all, it goes without saying that we don't want bugs appearing in the end result, right?

Tests can be performed manually, but there is an increased risk of the human factor, where we can't always be sure that the tests were performed 100 percent identically. To reduce (or even eliminate) this risk, we can automate tests using jQuery's unit testing suite, QUnit. We can, of course, run the QUnit tests manually, but the beauty of QUnit is that it can be completely automated, as we will see later in this chapter.

For now, let's take a moment to recap the basics of how to get QUnit installed and run some basic tests.

Installing QUnit

There are three ways to install QUnit. We can simply include the two links to it in our code, using the JavaScript and CSS files available at <https://qunitjs.com>. These can be referenced directly, as they are hosted on QUnit's CDN links that are provided by MaxCDN.

The alternative is to use NodeJS. To do this, we can browse to the NodeJS site at <http://www.nodejs.org>, download the appropriate version for our platform, and then run this command on the NodeJS command prompt:

```
npm install --save-dev qunitjs
```

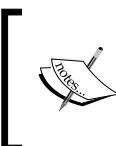
We can even use Bower to install QUnit; to do so, we need to first install NodeJS and then run this command to install Bower:

```
npm install -g bower
```

Once Bower is installed, QUnit can then be installed with this command:

```
bower install --save-dev qunit
```

At this stage, we're ready to start creating our automation tests with QUnit.



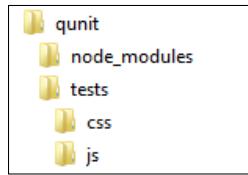
If you want to really push the boat out, you can test the latest committed version of QUnit—the links are available at <http://code.jquery.com/qunit/>; it should be noted that this is not for production use!



Creating a simple demo

Now that we have QUnit installed, we're ready to run a simple test. To prove that it's working, we're going to modify a simple demo in order to test for the number of letters in a textbox and indicate whether it is above or below a given limit, as follows:

1. We'll start by extracting copies of the code required for our demo from the code download that accompanies this book; go ahead and extract the `qunit.html` file along with the `css` and `js` folders and store these in your project area:



[ Don't worry about the presence of the `node_modules` folder; we will be creating this later in the chapter, when Node is installed.]

2. We now need to modify our test markup, so go ahead and open up `qunit.html` and then modify it, as indicated:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Testing jQuery With QUnit</title>
    <meta charset="utf-8">
    <link rel="stylesheet" href="css/qunit.css" />
    <link rel="stylesheet" href="css/qunittest.css" />
    <script src="js/jquery.min.js"></script>
    <script src="js/qunit.js"></script>
    <script src="js/qunittest.js"></script>
  </head>
  <body>
    <form id="form1">
      <input type="text" id="textLength">
      <span id="results"></span>
      <div id="qunit"></div>
      <div id="qunit-fixture"></div>
    </form>
  </body>
</html>
```

3. Next, open up a text editor of your choice and add the following code, saving it as `qunittest.js` in the `js` folder. The first block performs a check on the length of the text field and displays a count; it turns the background of that count red if it is over the prescribed length of eight characters:

```
$(document).ready(function() {
  var txt = $("input[id$=textLength]");
  var span = $("#results");
  $(txt).keyup(function() {
    var length = $(txt).val().length;
```

Testing jQuery

```
$(span).text(length + " characters long");
$(span).css("background-color", length >= 8 ?
  "#FF0000" : "#00FF00");
});
```

4. Add these lines of code immediately below the previous block; this calls QUnit to test for the length of our text field and displays the results below the letter count:

```
$(txt).val("Hello World!");
QUnit.test("Number of characters in text field is 8 or
more", function(assert) {
  $(txt).trigger("keyup");
  assert.ok($(txt).val().length >= 8, "There are " +
    $(txt).val().length + " characters.");
});
});
```

5. With the files in place, we're ready to run the tests; go ahead and run `qunit.html` in a browser. If all went well, we should see the results of our test, which in this instance will show a pass:

The screenshot shows a browser window with the following details:

- Page Title:** Testing jQuery With QUnit
- Test Result:** 1 assertions of 1 passed, 0 failed.
- Test Case:** 1. Number of characters in text field is 8 or more (1) Rerun
- Assertion:** 1. There are 12 characters. @ 4 ms

6. Not every test that we perform in real life will be successful; there will be occasions when our tests fail, if we've not provided the right values or performed a calculation that gives an unexpected result. To see how this looks in QUnit, go ahead and add these lines to the `qunittest.js` file, as shown here:

```

assert.ok($(txt).val().length >= 8, "There are " +
$(txt).val().length + " characters.");
});

$(txt).val("Hello World!");
QUnit.test("Number of characters in text field is 8 or less",
function(assert) {
    $(txt).trigger("keyup");
    assert.ok($(txt).val().length <= 8, "There are " +
    $(txt).val().length + " characters.");
});

```

7. Now, refresh your browser window; this time around, you should see the tests completed but with one failure, as shown in the following screenshot:

The screenshot shows the QUnit test runner interface. At the top, there's a status bar with a text input field containing "Hello World!" and a red button next to it labeled "12 characters long". Below this is a dark header bar with the text "Testing jQuery With QUnit". Underneath is a toolbar with several checkboxes: "Hide passed tests" (unchecked), "Check for Globals" (unchecked), "No try-catch" (unchecked), a "Filter" input field, and a "Go" button. The main content area has a blue header bar with the text "QUnit 1.18.0; Mozilla/5.0 (Windows NT 6.3; WOW64; rv:37.0) Gecko/20100101 Firefox/37.0". Below this, a message says "Tests completed in 16 milliseconds. 1 assertions of 2 passed, 1 failed." A list of tests follows:

- 1. Number of characters in text field is 8 or more (1)** Rerun 1 ms
 - 1. There are 12 characters. @ 1 ms
- 2. Number of characters in text field is 8 or less (1, 0, 1)** Rerun 9 ms
 - 1. There are 12 characters.

Expected: true
Result: false
Diff: trufalse
Source: @file:///C:/Users/alex/Desktop/Qunit%20Test/js/qunittest.js:19:9



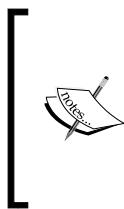
There is a completed version of this example in the code download, in the completed version folder, which explores the outcome of the tests.

Although this was designed as a simple demo, it nevertheless highlights how easy it is to create simple tests that give the appropriate responses; let's pause for a moment to consider what we have covered in this exercise.

The key to each test lies in the use of the `assert.ok()` function—this performs a simple Boolean check. In our examples, we check whether the text length is 8 characters or less or 8 characters or more and either show pass or fail depending on the outcome. Additionally, we can either ask QUnit to show a standard text or override it with a personalized message. This approach should be sufficient to get started with unit testing your code; as time progresses, we can always develop the tests further, if desired.

The beauty of this library is that we can use it with either jQuery or JavaScript; our examples in this chapter are naturally based around using the former, but QUnit is flexible enough to work with the latter, should we decide to move away from using jQuery in the future. QUnit is part of the jQuery family of products; there are similarities to other simple testing libraries, such as JUnit (available at <http://junit.org/>).

There is a huge amount that we can do when we harness the power of QUnit—what we've seen here only scratches the surface of what is possible to achieve.



If you want to learn more about the basics of QUnit, then I suggest that you refer to *Instant Testing with QUnit*, Dmitry Sheiko, available from Packt Publishing. There are lots of tutorials available online too; you can try this one, as a starting point: <http://code.tutsplus.com/tutorials/how-to-test-your-javascript-code-with-QUnit--net-9077>.

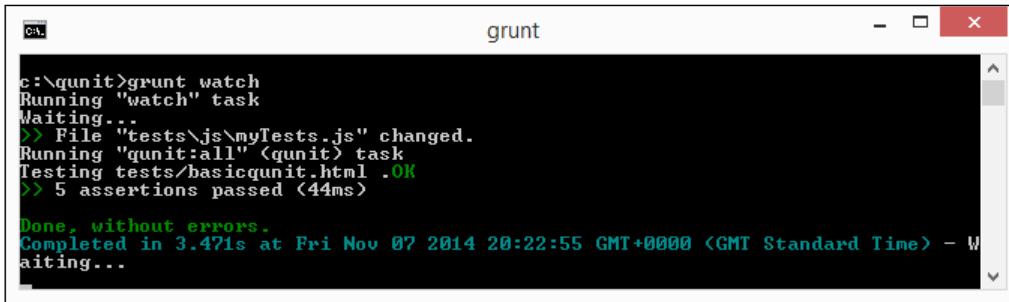
As a taster of what is possible, we will focus on one particular feature that will help you to take your jQuery development skills further: instead of running the tests manually each time, how about automating them completely so that they run automatically?

Automating tests with QUnit

Hold on, surely QUnit automates the running of these tests for us anyway?

The answer is yes and no. QUnit automated the tests but only to an extent; we had to run the set of tests manually each time. This is all well and good, but you know what? I'm feeling lazy and don't have the time or inclination to continually run tests by hand, as I am sure you won't either. We can do better than this; it is possible to automate our testing using NodeJS/Grunt and PhantomJS.

Granted, it will take some effort to set up, but it is worth the time saved when tests run automatically as soon as any identified content has been altered.



```
c:\qunit>grunt watch
Running "watch" task
Waiting...
>> File "tests\js\myTests.js" changed.
Running "qunit:all" <qunit> task
Testing tests/basicqunit.html .OK
>> 5 assertions passed <44ms>

Done, without errors.
Completed in 3.471s at Fri Nov 07 2014 20:22:55 GMT+0000 (GMT Standard Time) - Waiting...
```

Let's take a look at what is involved in automating our test:

1. We'll begin by installing NodeJS. To do this, browse to <http://nodejs.org/> and download the appropriate binary for your system; it is available for Windows, Mac OS, and Linux.
2. Once installed, go ahead and open up the NodeJS command prompt and then change to the qunit folder we created at the start of this chapter, in *Creating a simple demo*.
3. At command prompt, enter the following command:

```
npm install -g grunt-cli
```

NodeJS needs two files to be created in order to operate correctly; they are `package.json` and `gruntfile.js`. Let's go ahead and create them now.

4. Switch to a normal text editor of your choice and then in a new file, add the following code, saving it as `package.json`:

```
{
  "name": " projectName",
  "version": "1.0.0",
  "devDependencies": {
    "grunt": "~0.4.1",
    "grunt-contrib-QUnit": ">=0.2.1",
    "grunt-contrib-watch": ">=0.3.1"
  }
}
```

5. Revert to the NodeJS command prompt and then enter the following:

```
npm install
```

Testing jQuery

6. In a separate file, add the following code and save it as `gruntfile.js`:

```
module.exports = function(grunt) {
  grunt.initConfig({
    pkg: grunt.file.readJSON('package.json'),

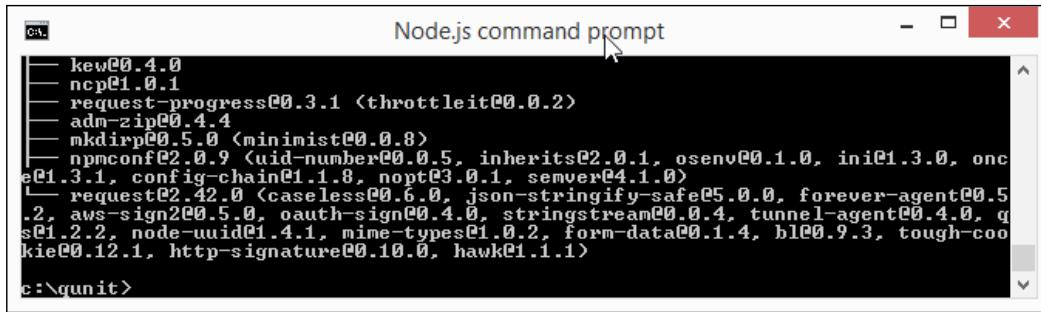
    QUnit: {
      all: ['tests/*.html']
    },
    watch: {
      files: ['tests/js/*.js', 'tests/*.html'],
      tasks: ['QUnit']
    }
  });

  grunt.loadNpmTasks('grunt-contrib-watch');
  grunt.loadNpmTasks('grunt-contrib-QUnit');
  grunt.registerTask('default', ['QUnit', 'watch']);
};
```

7. Revert to the NodeJS command prompt again and enter the following:

```
npm install -g phantomjs
```

8. If all went well, we should see something akin to the following screenshot appear:



9. Let's now start Grunt and set it to watch for any changes in our code; to do this, run this command in the NodeJS command prompt:

```
grunt watch
```

10. Open up a copy of `qunittest.js`, which we created earlier in this chapter, and then save the file—I know this might sound crazy, but it is required to trigger the process in Grunt.
11. If all went well, we should see this result appear in the NodeJS window:

```
c:\>qunit>grunt watch
Running "watch" task
Waiting...
>> File "tests\js\myTests.js" changed.
Running "qunit:all" <qunit> task
Testing tests/basicqunit.html .OK
>> 5 assertions passed (44ms)

Done, without errors.
Completed in 3.471s at Fri Nov 07 2014 20:22:55 GMT+0000 (GMT Standard Time) - Waiting...
```

12. Revert to `qunittest.js` and then change this line as shown here:

```
assert.ok($(txt).val().length <= 8, "There are " +
$(txt).val().length + " characters.");
```

13. Save your file and then watch the Grunt window, which should now indicate a failed test:

```
c:\>qunit>grunt watch
Running "watch" task
Waiting...
>> File "tests\js\myTests.js" changed.
Running "qunit:all" <qunit> task
Testing tests/basicqunit.html F
>> isEven()
>> Message: Three is an even number
>> Actual: null
>> Expected: undefined

Warning: 1/6 assertions failed (50ms) Use --force to continue.

Aborted due to warnings.
Completed in 3.497s at Fri Nov 07 2014 20:21:25 GMT+0000 (GMT Standard Time) - Waiting...
```

Let's change tack and move on to something else; although we've not covered the use of QUnit in any great depth, it is nevertheless important to try and follow best practices where possible, when using QUnit. Let's take a moment to consider some of these best practices in order to see how they can improve our coding skills.

Exploring best practices when using QUnit

The aim of any developer should be to follow best practices when possible; it's not always practical to do this, so it is important to learn where to compromise if circumstances dictate. Assuming this won't happen too often, there are a number of pointers we can try to follow, as a best practice, when using QUnit:

- **Make each test independent of each other:** Each test we run should only test one specific behavior; if we test the same behavior in multiple tests, then we will have to change all the tests if the behavior needs to be changed.
- **Don't make unnecessary assertions:** Ask yourself this question, "What behavior are we trying to test?". A unit test is meant to be a design schematic of how a certain behavior should work and not detail everything the code happens to do. Try to keep assertions to one per test where possible; there is no point in running a test if an assertion is already tested elsewhere in our code.
- **Test only one code unit at a time:** The architecture design of your code must support testing units (that is, classes or very small groups of classes) independently and not chained together. If not, you risk creating a lot of overlap, which will cascade and cause failures elsewhere in your code. If the design of your application or site doesn't allow this, then the quality of your code will suffer; it may be necessary to use **Inversion of Control (IoC)** to test your work instead.



The usual practice is for the custom code to call into generic, reusable libraries (such as QUnit); IoC flips the process so that in this instance, the tests are performed by QUnit calling our custom code instead.



- **Mock out all external services and state data:** A key part of unit testing is to reduce the effect of external services on your code, where possible—the behavior of these services can overlap with your tests and impact the results.

- **Avoid mocking up too many objects or state data:** If you have any data that controls the state of your application or site, then try to keep any mock data to below 5 percent; anything higher and you risk making your tests less trustworthy. It is also wise to reset these back to a known value before running successive tests, as different tests can influence these values for other tests. If you find that you have to run tests in a specific order or that you have a dependency on an active database or network connections, then your design or code is not right and you should revisit both to understand why and how the dependencies can be removed.
- **Avoid unnecessary preconditions:** Avoid having common setup code that runs at the beginning of lots of unrelated tests. This will confuse your tests, as it won't be clear which assumptions your test relies on and indicates that you're not testing just a single unit. It is key to create the right conditions, even though this can be difficult—the trick is to keep them simple as much as it is practical.
- **Don't unit test configuration settings:** There is no benefit in inspecting your configuration settings when running unit tests; it will likely result in duplicated code, which isn't necessary.
- **Don't specify your implementation – specify the result instead:** Unit testing is designed to focus on the results, not the implementation—does your function produce what you expect it to do? Take an example of the following code snippet:

```
test("adds user in memory", function() {
  var userMgr = makeUserMgr();
  userMgr.addUser("user", 'pass');
  equal(userMgr._internalUsers[0].name, "user")
  equal(userMgr._internalUsers[0].pass, "pass")
});
```

This seems perfectly reasonable, right? It's perfectly valid, if it weren't for the fact that it focuses on *how* the code was implemented, not the result.

A better way to test the code is to use this approach instead:

```
test("adds user in memory", function() {
  var userMgr = makeUserMgr();
  userMgr.addUser("user", "pass");
  ok(userMgr.loginUser("user", "pass"));
});
```

In this example, we're not focusing on the route taken to get our result but on the end result itself; does it produce what we need to see?

- **Name your unit tests clearly and consistently:** A successful unit test will clearly indicate its purpose; a useful way to name tests is using what I call the **SSR** principle, or **Subject, Scenario, and Result**. This means that we can identify what is being tested, when the tests should be run, and what the expected result will be. If we simply name it with just the subject, then it will become difficult to maintain if we don't know what we're trying to maintain!

These tips only scratch the surface of what should be followed as a good practice; for a more in-depth discussion, it is worthwhile to read the article by Adam Kolawa on applying unit testing, which is available at <http://www.parasoft.com/wp-content/uploads/pdf/unittesting.pdf>. The key point to remember though is to keep it simple, logical, and not try to over-complicate your tests, or they will become meaningless!

Summary

We've now reached the end of the chapter; even though it was short, it covered some useful points on unit testing practices and how we can save ourselves time and effort by automating our testing. Let's quickly recap what we learned.

We kicked off with a quick revisit of the principles of QUnit and how to install it; we briefly covered the most popular method but also looked at how to use CDN and Bower to make the libraries available in our code.

Next up came a look at some basic examples of testing; while these are very simple, they highlighted the principles we should use in unit testing. These were explored further with a discussion on the best practices to follow when unit testing with QUnit.

We've now reached the end of the book. I hope you enjoyed our journey through *Mastering jQuery* and have seen that it is not just about writing code but about some of the more soft topics, which will help enhance your skills as a jQuery developer.

Index

Symbols

- 7-Zip**
 - URL 297
- \$.proxy function**
 - reference link 179
 - using 177-179
- .stop() function**
 - reference link, for documentation 137

A

- Abstract Pattern** 68
- Adapter Pattern**
 - about 51
 - advantages 51
 - drawbacks 51
- advanced contact form**
 - creating, AJAX used 99-101
 - modifying 122-125
- advanced file upload form**
 - developing, jQuery used 102-104
- advanced plugin**
 - designing 271
- AJAX**
 - best practices 128, 129
 - defining 109
 - history 108
 - used, for adding file upload capabilities 125, 126
 - used, for creating advanced contact form 99-101
 - used, for creating simple example 110-112
- AJAX content**
 - caching, localStorage used 115-117

- ajax() object**
 - reference link 111
- Anima.js**
 - reference link 144
- animate.css library**
 - reference link 203
- animate() method**
 - exploring 191, 192
- Animation Cheat Sheet**
 - reference link 155
- animation performance**
 - considering, on responsive sites 152, 153
- animation requests**
 - handling, on responsive site 154-156
- animations**
 - appearance, improving of 159, 160
 - converting, for automated CSS usage 276, 277
 - references 140, 346
 - versus effects 190
- animations, and transitions affect**
 - performance**
 - reference link 160
- animation support, jQuery**
 - about 29
 - demo, creating 30-33
- Animo.JS**
 - reference link 144
- application**
 - building 299, 300
 - debugging 310
 - deploying 310-315
 - installing 299, 300
 - packaging 310

reference link, for packaging and distributing 310

apply() function

reference link 193

Asynchronous Module Definition (AMD)

about 9

used, for loading jQuery 9-11

automated perf optimization

references 346

B

basic form

creating 75, 76

behavioral patterns 49

behaviors

modifying 23

replacing 23

Be Moved

URL 164

BenchmarkJS

URL 323

best practices, AJAX 128, 129

best practices, QUnit 360, 361

Bezier curve

reference link 144

support, adding 200-202

using, in effects 199

bezier-easing

reference link 281

Bez plugin

references 143, 200, 279

Blender Foundation

URL 220

BlueImp plugin configuration

dissecting 305-308

boilerplate

used, for rebuilding plugin 272-276

Bootstrap

URL 272

Bower

URL 43

used, for installing jQuery 6-8

used, for packaging plugin 283-285

Builder Pattern

about 67, 68

advantages 68

disadvantages 69

C

Calibreapp

URL 323

callback hell

about 118

reference link 118

callbacks

adding, to effects 204

reference link 58

used, for handling multiple AJAX

requests 117, 118

CamanJS

about 244

creative example 246, 247

filters, applying with 244

reference link, for example 248

simple demo, building 244-246

URL 244

Can I Use

reference link 257

CDN

links 4

URL 244

using 4

Ceaser

reference link 140

clickToggle handler

creating 192-194

code

enhancing, with Deferreds 118-120

enhancing, with Promises 118-120

minifying, NodeJS used 335-337

CodeCanyon

reference link 248

CodePen

reference link 227

Codrops demo 145

Colorimazer

URL 248

colors

manipulating, in images 235, 236

CommonJS 6
Complete Widget Factory 270
Composite Pattern
 about 49, 50
 advantages 50
 drawbacks 50
configuration options, \$.ajax object
 data 109
 dataType 109
 error 109
 reference link 110
 Success 109
 type 109
 url 109
configuration options, AJAX-enabled code
 cache 110
 data 110
 datatype 110
 jsonp 111
 statusCode 111
 type 110
 url 110
consistent code style, jQuery
 reference link 289
content
 animating, for mobile devices 157, 158
 controlling, with jQuery's Promises 205-207
 sliding, with slide-fade Toggle 194-196
Content Delivery Network (CDN) 1
content files
 Bluelmp plugin configuration,
 dissecting 305-308
 dissecting 304
 project creation, automating 308, 309
 window.js, exploring 305
Core library
 patching, on run 22
Costa Coffee
 URL 164
creational patterns 49
Cross-Origin Resource Sharing (CORS) 108
CSS
 selecting 132-134
CSS3
 reference link 204
 used, for adding filters 236, 237
 used, for blending images 242, 243
CSS3 styling
 using, considerations 279, 280
CSS Animate
 reference link 156
cssAnimate library
 about 255-257
 URL, for downloading 255
CSS-based animations
 working with 278
CSS Media Queries boilerplate
 reference link 155
CSS specificity
 reference link 153
CSS-Tricks
 reference link 151
CSS version
 references 165
cubic-bezier
 reference link 257
custom animations
 designing 141-143
 implementing 145
 reference link 144
custom easing
 adding, to effects 197-199
Custom Easing Function Explorer
 URL 198
custom easing functions
 applying, to effects 196, 197
 references 197
custom effects
 creating 190
 working 192
custom events
 creating 181-183
 reference link 180
custom event types
 creating 180
 decoupling 180
custom validators
 URL, for documentation 93
custom version, of Modernizr
 URL 99

D

Deferreds

advantages 120
code, enhancing with 118-120
usage, demonstrating 127, 128
working with 120, 121

delegated events

reference link 129

descendant anchors 170

design patterns

about 46, 269
behavioral patterns 49
benefits 46
categorizing 49
creational patterns 49
defining 46, 47
elements 48
features 47
relating, to jQuery 69-71
structural design patterns 49
structure, dissecting 48

desktop

HTML applications, operating on 295-297

development environment

preparing 297, 298

DeviceTiming

URL 323

Document Object Model (DOM) 109

Don't Repeat Yourself (DRY) principles 47

Drupal files

minifying, example 338

duck punching 22

Dynamic Favicons library

URL 220

E

easeInQuint

references 279

easeOutQuint

references 143

easing functions

about 140

references 140, 280

effect queue

creating 207-209

managing 207-209

effects

about 189
Bezier curves, using in 199
callbacks, adding to 204
custom easing, adding to 197-199
custom easing functions,
 applying to 196, 197
versus animation 190

e-mails

regex validation function, creating for 82

ETags

URL 114

event capturing

reference link 173

event delegation

about 168
basics 168, 169
code, reworking 170
controlling 173-175
controlling, stopPropagation() method
 used 175, 176

demonstration, exploring 171, 172

implications, exploring 172, 173

older browsers, supporting 170

references 170, 176

event handling 167

event propagation 169

Eventralize library

reference link 187

events

namespace, adding to 184-186

Extensible Stylesheet Language

Transformations (XSLT) 109

F

Facade Pattern

about 52
advantages 54
drawbacks 55
simple animation, creating 53, 54

files compression

tips 337, 338

files, Node-WebKit folder

content files, dissecting 304
ffmpegsumo.dll 304
filesizeview.nw 304

gruntfile.js 304
icudtl.dll 304
libEGL.dll 304
libGLESv2.dll 304
nw.exe 304
nw.pak 304
package.json 304

file upload capabilities
adding, AJAX used 125, 126

filters
applying, with CamanJS 244
contrast() 241
exploring 241
grayscale() 241
hue-rotate() 241
images, animating with 255
invert() 242
reference link 242, 243
reference link, for article 242
Saturate() 242
simple filters, creating manually 248

filters, adding with CSS3
about 236, 237
base page, creating 237-239
brightness level, modifying 240
sepia filter, adding to image 240, 241

Firebug
speed of jQuery, monitoring with 321-323
URL 321

FireQuery Reloaded
URL 323

FireStorage Plus! plugin
URL 116

fixHooks 180

Font Squirrel
references 9, 141, 150

forms
validating, jQuery used 79, 80
validating, regex statements used 81

form validation
common errors 74
key elements 74
need for 74

Form Validator
URL 89

G

Gang of Four (GoF) 46
getImageData() method
reference link 253
getUserMedia.js
reference link 264

Gist
URL 182

Google PageSpeed
URL 330
used, for gaining insight 330-332

Graphics Processing Unit (GPU) 157

grunt-bump
reference link 285

grunt-contrib-jshint
reference link 333

grunt-contrib-watch
reference link 311

grunt-node-webkit-builder
reference link 311

grunt-topcoat-telemetry
reference link 332

grunt-yslow
reference link 324

GUI
using 14

gulp-grunt
reference link 332

H

hardware acceleration, and CSS3
reference link, for impact 133

hoverFlow plugin
reference link 137

Hoverizr
URL 248

HTML5
versus jQuery 78

HTML5 Boilerplate
URL 272

HTML5 validation
performing 76-78

HTML applications
operating, on desktop 295-297

I

IIFEs

reference link 64

images

animating, with filters 255
blending 253, 254
blending, CSS3 used 242, 243
colors, manipulating in 235, 236
exporting 258-260
grayscale 249, 250
sepia filter, adding to 240, 241

ImagesLoaded

URL 162

Immediately Invoked Function Expression (IIFE) 23

implications

considering, of parallax scrolling 164, 165

Inno Setup

URL, for downloading 314

Inversion of Control (IoC) 360

iScroll.js

URL 165

Iterator Pattern

about 58-60
advantages 60
disadvantages 60

J

Jam

URL 43

jQuery

about 21
animation support, updating in 29
CDN, using 4
design patterns, relating to 69-71
downloading 1, 2
installing 1-4
installing, Bower used 6-8
installing, NodeJS used 5
jQuery Migrate plugin, adding 3
loading, AMD used 9-11
loading, best practices 18, 19
reference link 170
reference link, for parsing and execution 346

selecting 132-134

URL 2

used, for developing advanced file upload form 102-104

used, for validating forms 79, 80
used, in development capacity 2
uses 349, 350
versus HTML5 78

jQuery Animate Enhanced

URL 145

jQuery animation queue

controlling 134, 135
problem, fixing 135, 136
pure CSS solution, using 138, 139
transition, making smoother 136, 137

jQuery animations

about 280-282
improving 139, 140

jQuery Boilerplate templates

URL 272

jQuery code

linting, automatically 332-335

jQuery Color

URL 192

jQuery downloads

customizing, from Git 11
GUI, using as alternative 14
redundant modules, removing 13

jQuery Easing

reference link 141

jQuery.fx.interval

reference link 159

jQuery.Keyframes plugin

reference link 144

jquery-lazy

reference link 345

jQuery Learning Site

reference link 183

jQuery Migrate plugin

adding 3

jquery.min.js

reference link 150

jQuery object

jQuery.migrateMute 3

jQuery.migrateReset() 3

jQuery.migrateTrace 3

jQuery.migrateWarnings 3

jQuery Patch
reference link 146
jquery.smoothstate.js
reference link 150
jQuery's Promises
content, controlling with 205-207
jQuery Timer Tools
URL 153
jQuery UI
URL 192
JSDoc
URL 285
JSFiddle
URL 160
JSHint
URL 289
JSLint
URL 289
jsLitmus
URL 323
JSPerf
URL 343
JSPerf.com
URL 323
JUnit
URL 356

K

KISS principle 74

L

Laplace filter
reference link 255
lazy initialization pattern
about 61
advantages 61
drawbacks 62
lazy loading plugin
reference link 61
Learn jQuery site
reference link 290
Lightweight Start 270
localStorage
reference link 115
used, for caching AJAX content 115-117

M

MAMP
URL 99
MaxCDN 4
Mini AJAX File Upload Form
reference link 302
mobile devices
content, animating for 157, 158
Modernizr
about 17
URL 17
working with 16
modules
reference link 305
modules, regex examples
reference link 93
monkey patch
creating 24, 25
dissecting 26-28
monkey patching
about 22
benefits 28
pitfalls 41
Multiclick event plugin
URL 183
working with 183, 184
multiple AJAX requests
handling, callbacks used 117, 118

N

namespace
adding, to events 184-186
Node
URL 298
NodeJS
URL 284
used, for installing jQuery 5
used, for minifying code 335-337
Node Package Manager (NPM) 5
Node-WebKit
about 294, 295
references, for resources 316, 317
Node-WebKit manifest file
reference link 301

NPM
 URL 323
nw.js
 URL 294

O

Observer Pattern
 about 55
 advantages 56
 basic example, creating 57, 58
 drawbacks 57
open/closed principle
 reference link 65
overlay effect
 animating 147-149
 reference link 148

P

package creation process
 automating 311-314
package.json file
 dissecting 301
 reference link 301
packages
 creating, manually 311
packaging content for download, via Bower
 reference link 43
Page Visibility API
 about 211, 212
 audio, pausing 220, 221
 implementing 212, 213
 properties 214
 references, for examples 223, 224
 support, adding to CMS 221-223
 support, detecting 214-216
 supporting 212
 using, in practical context 220
 video, pausing 220, 221
PaintbrushJS
 URL 248
parallax.js plugin, PixelCog
 reference link 161
parallax scrolling
 implications, considering of 164, 165
parallax scrolling page
 building 161-164
 reference link 161
parallax scrolling responsive
 reference link 164
parallax scrolling websites
 reference link 165
patches
 applying 42, 43
 distributing 42, 43
patterns
 creating 270
 using 270
performance
 about 320, 321
 best practices, implementing 342-346
 strategy, designing 347, 348
performance monitoring
 automating 324-329
Phonestagram application
 reference link 246
plugin
 best practices and principles 289, 290
 extending 282, 283
 packaging, Bower used 283-285
 rebuilding, boilerplate used 272-276
 values, returning from 286-289
plugin architecture for validation, developing
 about 88
 basic form, creating 89, 90
 content, localizing 94
 custom validators, creating 90-93
 development, wrapping up 96, 97
 error messages, centralizing 95, 96
 fallback support, providing 98
 usage of best practices, noting 97, 98
plugin design patterns
 reference link 270
plugin example, JSFiddle
 reference link 273
plugin pattern
 reference link 271
plugin pattern, key aspects
 architecture 270
 maintainability 270
 reusability 271

poorly developed plugin
signs, detecting of 267-269
premature optimization 320
Promises
advantages 120
code, enhancing with 118-120
usage, demonstrating 127, 128
working with 120, 121
provision of documentation
automating 285, 286
Proxy Pattern
about 65
advantages 66, 67
disadvantages 67
publish/subscribe model
reference link 56
pure CSS
solution, using 138, 139
using 202, 203

Q

QTransform
URL 145
QUnit
about 351
best practices 360, 361
installing 352
simple demo, creating 352-356
tests, automating with 356-360
URL 352

R

redundant modules
removing 13
regexes
URL, for blog 91
regex statements
used, for validating forms 81
regex validation function
creating, for e-mails 82
requestAnimationFrame (rAF)
about 29, 224, 225
changes, retrofitting to jQuery 227, 228
examples 229

exploring 30
Google Maps marker, animating 231, 232
references 30, 227
resources, exploring 232, 233
scrollable effect, creating 229, 230
using 30, 226
working 226
responsive parallax scrolling
implementing 161
responsive sites
animating in 149-151
animation performance,
considering on 152, 153
animation requests, handling on 154-156
Reveal.js library
reference link 264
rollover buttons
animating 145, 146
code, exploring 146, 147

S

scene
setting 294
Scrollability plugin
URL 165
security page, nw.js
URL, for wiki 297
sepia filter
adding, to image 240, 241
sepia tone
adding 251, 252
signature pad
creating 258-260
Signature Pad plugin
reference link 258
signs
detecting, of poorly developed
plugin 267-269
simple application
building 302, 303
demo, exploring 304
simple filters
creating, manually 248
simple validation plugin
building 84-88

Single Page Application (SPA) 149
Skrollr
 references 161, 162
slide-fade Toggle
 content, sliding with 194-196
Sobel filter
 reference link 255
source maps
 support, adding 16
 URL 16
SpeedCurve
 URL 323
speed, of jQuery
 monitoring, with Firebug 321-323
speed, of loading data
 improving, with static sites 113, 114
SSR principle 362
Stack Overflow
 URL 170
static sites
 speed of loading data, improving
 with 113, 114
Stellar.js
 reference link 161
stopPropagation() method
 used, for controlling event
 delegation 175, 176
Strategy Pattern
 about 62
 actions, switching between 64
 advantages 64
 disadvantages 65
 simple toggle effect, building 63, 64
structural design patterns 49
sublime-grunt
 reference link 336
Sublime Text
 URL, for downloading 297
Sublime Text 2
 reference link 310
support, Page Visibility API
 demo, building 218, 219
 detecting 214-216
 fallback support, providing 217
 visibility.js, installing 217
Syntactically Awesome Stylesheets (SASS) 155

T

Team Treehouse
 reference link 227
Test Mail Server tool
 references 99
tests
 automating, with QUnit 356-360
Thumbelina plugin
 reference link 229
toDataURL() method
 reference link 260
Toggles plugin
 reference link 194
transform
 reference link 147
transitionend, Treehouse website
 reference link 140

U

UglifyJS
 URL 336
unit testing
 URL 362
unused JavaScript
 working out 339-341
URL validation
 performing 83

V

values
 returning, from plugin 286-289
Velocity.js
 about 139
 references 139, 154
ViewPortSize
 URL 162
VintageJS
 URL 248
visibility.js
 installing 217
 references 211, 217

W

- WAMP**
 - URL 99
- Web Animations API**
 - references 134
- webcam images**
 - capturing 260-264
 - manipulating 260-264
- Web Graphics Library (WebGL)** 304
- WebP** 34
- WebP support, adding to jQuery**
 - about 34, 35
 - patch, adding 35-39
 - working 39, 40
- window.js**
 - exploring 305
 - reference link 305
- WOW Slider**
 - URL 61

X

- XAMPP**
 - URL 99
- Y**
- Yeoman Node WebKit Generator**
 - URL 308
- YSlow**
 - references 326



Thank you for buying Mastering jQuery

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.



jQuery for Designers Beginner's Guide

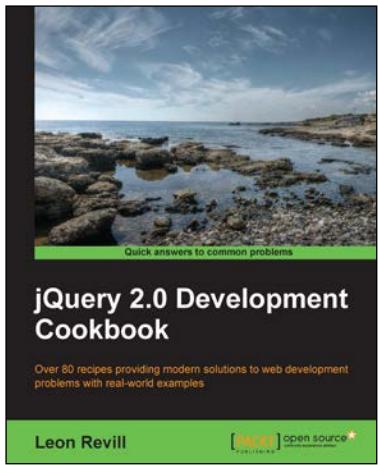
Second Edition

ISBN: 978-1-78328-453-5

Paperback: 398 pages

Design interactive websites to improve user experience by using the popular JavaScript library

1. Enhance the user experience of your site by adding useful jQuery features – provide easy navigation, communicate updates and changes, and allow site visitors to interact with content.
2. Learn the modular approach to jQuery, including the addition of plug-ins to achieve advanced effects without writing much code at all.



jQuery 2.0 Development Cookbook

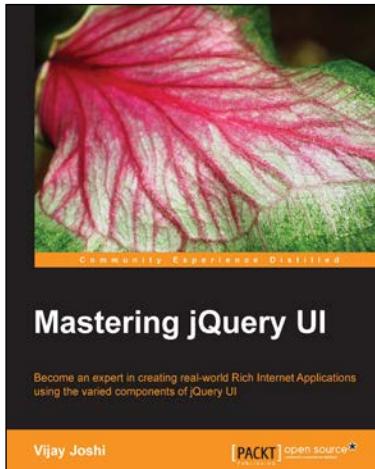
ISBN: 978-1-78328-089-6

Paperback: 410 pages

Over 80 recipes providing modern solutions to web development problems with real-world examples

1. Create solutions for common problems using best practice techniques.
2. Harness the power of jQuery to create better websites and web applications.
3. Break away from boring websites and create truly intuitive websites and web apps, including mobile apps.

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

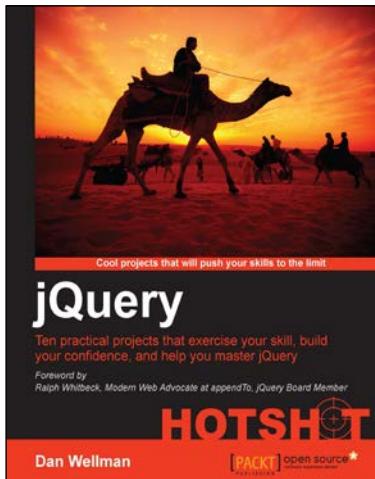


Mastering jQuery UI

ISBN: 978-1-78328-665-2 Paperback: 312 pages

Become an expert in creating real-world Rich Internet Applications using the varied components of jQuery UI

1. Create useful mashups by plugging together different components along with APIs.
2. Design your own widgets like captchas, a color picker, news reader, puzzles, and many others.
3. Take your jQuery UI skills to next level by exploring the ins and outs and nuances of jQuery UI components.



jQuery HOTSHOT

ISBN: 978-1-84951-910-6 Paperback: 296 pages

Ten practical projects that exercise your skill, build your confidence, and help you master jQuery

1. See how many of jQuery's methods and properties are used in real situations. Covers jQuery 1.9.
2. Learn to build jQuery from source files, write jQuery plugins, and use jQuery UI and jQuery Mobile.
3. Familiarise yourself with the latest related technologies like HTML5, CSS3, and frameworks like Knockout.js.

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