



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 JavaScript High Performance

Master the art of building, deploying, and optimizing faster web applications with JavaScript

Chad R. Adams

[PACKT] open source*
PUBLISHING community experience distilled

Mastering JavaScript High Performance

Master the art of building, deploying, and optimizing
faster web applications with JavaScript

Chad R. Adams



BIRMINGHAM - MUMBAI

Mastering JavaScript High Performance

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: March 2015

Production reference: 1250315

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

ISBN 978-1-78439-729-6

www.packtpub.com

Credits

Author

Chad R. Adams

Project Coordinator

Milton Dsouza

Reviewers

Yaroslav Bigus

Andrea Chiarelli

Vishal Rajpal

Proofreaders

Stephen Copestake

Paul Hindle

Indexer

Tejal Soni

Commissioning Editor

Ashwin Nair

Production Coordinator

Melwyn D'sa

Acquisition Editor

Owen Roberts

Cover Work

Melwyn D'sa

Content Development Editor

Parita Khedekar

Technical Editor

Anushree Arun Tendulkar

Copy Editors

Hiral Bhat

Vikrant Phadke

Stuti Srivastava

About the Author

Chad R. Adams is a mobile frontend architect, currently working at Intouch Solutions, where he looks at creative ways of building HTML5-driven content and native iOS, Android / Windows Runtime applications. He lives in Raymore, Missouri, with his wife, Heather, and son, Leo.

In the past, Chad worked as a web developer for large websites, such as `MSN.com`, `Ford.ca`, `Xbox.com`, `WindowsPhone.com`, and `Copia.com`. He also speaks at developer conferences and groups in the Kansas City area on HTML5 and mobile development and is the author of *Learning Python Data Visualization*, Packt Publishing.

You can contact Chad on LinkedIn (<http://www.linkedin.com/in/chadradams>), Twitter (@chadradams), or his website (<http://chadradams.com>).

About the Reviewers

Yaroslav Bigus is an expert in building cross-platform web and mobile applications. He has over 5 years' experience in development and has worked for companies in Leeds and New York. He has used the .NET Framework stack to develop backend systems, JavaScript, AngularJS, jQuery, Underscore for frontends, and Xamarin for mobile devices.

Yaroslav is working for an Israeli start-up called Tangiblee. He has reviewed *Xamarin Mobile Application Development for iOS*, Packt Publishing, written by Paul F. Johnson; *iOS Development with Xamarin Cookbook*, Packt Publishing, written by Dimitris Tavlikos; and *Learning JavaScript Data Structures and Algorithms*, Packt Publishing, written by Loiane Groner.

I am thankful to my friends and family for their support and love.

Andrea Chiarelli has over 20 years' experience as a software engineer and technical writer. In his professional career, he has used various technologies for the projects he was involved in, from C#, JavaScript, and ASP.NET to AngularJS, REST, and PhoneGap/Cordova.

Andrea has contributed to many online and offline magazines, such as *Computer Programming* and *ASP Today*, and coauthored a few books published by Wrox Press.

Currently, Andrea is a senior software engineer at the Italian office of Apparound, a mobile software company founded in the heart of Silicon Valley. He is a regular contributor to <http://www.HTML.it>, an Italian online magazine focused on web technologies.

Vishal Rajpal is an experienced software engineer who started developing professional software applications in 2011. He has worked primarily on Java, Javascript, and multiplatform mobile application development, including PhoneGap and Titanium.

Vishal is pursuing his master's degree in computer science from Northeastern University, Seattle, and has worked on C, Java, and JavaScript. He lives in Seattle and can be contacted at vishalarajpal@gmail.com. You can also read more about his work at <https://github.com/vishalrajpal/> and <http://www.vishal-rajpal.blogspot.in>.

Vishal has also worked on books by Packt Publishing, such as *PhoneGap 3.x Mobile Application Development HOTSHOT* and *Learning Javascript, Data Structures, and Algorithms*.

www.PacktPub.com

Support files, eBooks, discount offers, and more

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

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

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



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

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

Why subscribe?

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

Free access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface	v
Chapter 1: The Need for Speed	1
Weren't websites always fast?	1
Getting faster	3
Selecting an effective editor	3
Integrated Development Environments	4
Mid-range editors	8
Lightweight editors	10
Cloud-based editors	12
Summary	15
Chapter 2: Increasing Code Performance with JSLint	17
Checking the JavaScript code performance	17
About the console.time API	18
When to use console.time	21
What is JavaScript linting?	22
About JSLint	22
Using JSLint	24
Reviewing errors	26
Configuring messy white space	27
Understanding the use strict statement	30
Using console in JSLint	32
Summary	34
Chapter 3: Understanding JavaScript Build Systems	35
What is a build system?	35
Compiling code by example	36
Error-checking in a JavaScript build system	37
Adding optimization beyond coding standards	38

Creating a build system from scratch using Gulp.js	38
Node.js	39
Setting up our build system	47
About Grunt.js and Gulp.js	47
Grunt Task Runner	48
About Gulp	48
Integrating JSLint into Gulp	53
Testing our example file	54
Creating a distribution	56
Summary	58
Chapter 4: Detecting Performance	59
Web Inspectors in general	59
The Safari Web Inspector	60
Firefox Developer tools	60
Internet Explorer developer tools	61
Chrome's Developer tools	62
Getting familiar with Chrome's Developer tools	64
Summary	82
Chapter 5: Operators, Loops, and Timers	83
Operators	84
The comparison operator	84
Is strict faster?	84
Loops	86
How loops affect performance	86
The reverse loop performance myth	89
Timers	92
What are timers and how do they affect performance?	93
Working around single-threading	94
Closing the loop	96
Summary	96
Chapter 6: Constructors, Prototypes, and Arrays	97
Building with constructors and instance functions	98
A quick word	98
The care and feeding of function names	98
Understanding instances	101
Creating instances with 'new'	101
Alternate constructor functions using prototypes	106
Understanding prototypes in terms of memory	106
Which is faster, a prototype or a constructor function?	107

Array performance	108
Optimizing array searches	109
Summary	112
Chapter 7: Hands off the DOM	113
<hr/>	
Why worry about the DOM?	113
Don't we need an MV-whatever library?	114
Creating new objects using the createElement function	115
Working around the createElement function	115
Working with the createElement function	116
When to use the createElement function?	120
Animating elements	120
Animating the old-fashioned way	120
Animating using CSS3	122
An unfair performance advantage	124
Understanding paint events	126
How to check for paint events?	126
Testing paint events	128
Pesky mouse scrolling events	129
Summary	132
Chapter 8: Web Workers and Promises	133
<hr/>	
Understanding the limitations first	133
Web workers	134
Testing workers with a local server	140
Promises	142
Testing a true asynchronous promise	144
Summary	147
Chapter 9: Optimizing JavaScript for iOS Hybrid Apps	149
<hr/>	
Getting ready for iOS development	149
iOS hybrid development	150
Setting up a simple iOS hybrid app	153
Using Safari Web Inspector for JavaScript performance	158
Comparing UIWebView with Mobile Safari	161
Common ways to improve hybrid performance	163
The WKWebView framework	166
Summary	167

Chapter 10: Application Performance Testing	169
What is unit testing in JavaScript?	170
Unit testing with Jasmine	170
Installation and configuration	171
Reviewing the project code base	173
Reviewing an application's spec for writing tests	175
Writing tests using Jasmine	177
Fixing our code	179
Summary	182
Index	183

Preface

Welcome to *Mastering JavaScript High Performance*. In this book, we have covered JavaScript performance in a way that helps any JavaScript developer, whether they are new to the language or are experienced veterans. This book covers common performance bottlenecks, how to look for performance issues within code, and how to correct them easily.

We also review modern ways of optimizing our JavaScript code not just by relying on sheer knowledge of JavaScript, but by using tools to help optimize code for us. These tools include Gulp and Node.js, which help create great performing builds, and Jasmine, a JavaScript unit-testing framework that helps discover application flow issues in JavaScript. We also debug a hybrid app using Apple Xcode debugging tools for HTML and JavaScript.

What this book covers

Chapter 1, The Need for Speed, explains the need for faster JavaScript, discusses why JavaScript code is traditionally slow, and shows the types of code editors that can help us write faster JavaScript, without changing our coding style.

Chapter 2, Increasing Code Performance with JSLint, explores performance fixes in JavaScript, and covers JSLint, a very good JavaScript validation and optimization tool.

Chapter 3, Understanding JavaScript Build Systems, teaches you JavaScript build systems and their advantages for JavaScript performance testing and deployment.

Chapter 4, Detecting Performance, covers Google's Development Tools options and contains a review of how to use a Web Inspector to improve our JavaScript's code performance.

Chapter 5, Operators, Loops, and Timers, explains operators, loops, and timers in the JavaScript language and shows their effect on performance.

Chapter 6, Constructors, Prototypes, and Arrays, covers constructors, prototypes, and arrays in the JavaScript language and shows their effect on performance.

Chapter 7, Hands off the DOM, contains a review of the DOM in relation to writing high-performance JavaScript, and shows how to optimize our JavaScript to render our web applications visibly faster. We also take a look at JavaScript animation and test performance against modern CSS3 animation.

Chapter 8, Web Workers and Promises, demonstrates web workers and promises. This chapter also shows you how to use them, including their limitations.

Chapter 9, Optimizing JavaScript for iOS Hybrid Apps, covers optimizing JavaScript for mobile iOS web apps, (also known as hybrid apps). Also, we take a look at the Apple Web Inspector and see how to use it for iOS development.

Chapter 10, Application Performance Testing, introduces Jasmine, a JavaScript testing framework that allows us to unit-test our JavaScript code.

What you need for this book

For this book, you will need a basic understanding of JavaScript, how to write functions and variables in JavaScript, how to use basic web technologies such as HTML and CSS, as well as some basic debugging skills using a Web Inspector such as Chrome Developer tools or Firebug, to name a few.

You will need a text editor, preferably for HTML and JavaScript coding; the available choices are covered in *Chapter 1, The Need for Speed*. Choosing the editor and the admin rights to the system you're working on is up to you, and it also depends on your budget. Also, *Chapter 9, Optimizing JavaScript for iOS Hybrid Apps*, strictly covers JavaScript in iOS development; for that, you will need a copy of Xcode and an Intel-based Mac. If you don't have these, you can still read along but, ideally, most of this work is done with a Mac.

Who this book is for

This book is written for intermediate JavaScript developers. If you are experienced with unit-testing JavaScript and writing your own frameworks, and are able to understand what instance-based versus static-based is in JavaScript, this book may not be for you. Also, if you're very new to JavaScript – as in, "How do I use a function?" – I recommend looking for a beginner's JavaScript book as well.

However, if you've been into JavaScript for a while but are new to node-style performance testing, grunt or gulp project deployments, and unit-testing in JavaScript, or if you want to know more on how to write JavaScript faster, or if you're just looking to stop your code base from lagging behind without reworking your coding style, you are reading the right book.

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 explanations of their meanings.


Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "To solve this issue, modern browsers have implemented new console functions called `console.time` and `console.timeEnd`."

A block of code is set as follows:

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

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: "Clicking on the **Next** button moves you to the next screen."

 Warnings or important notes appear in a box like this.

 Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book – what you liked or 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 topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on www.packtpub.com/authors.

Customer support

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

Downloading the example code

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

Downloading the color images of this book

We also provide you a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from: https://www.packtpub.com/sites/default/files/downloads/72960S_ColorImages.pdf.

Errata

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

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

The Need for Speed

In this chapter, we will learn about the need for executing JavaScript more rapidly, discuss why JavaScript code is traditionally slow, and see what kind of code editors can make us write faster JavaScript without changing our coding style.

Weren't websites always fast?

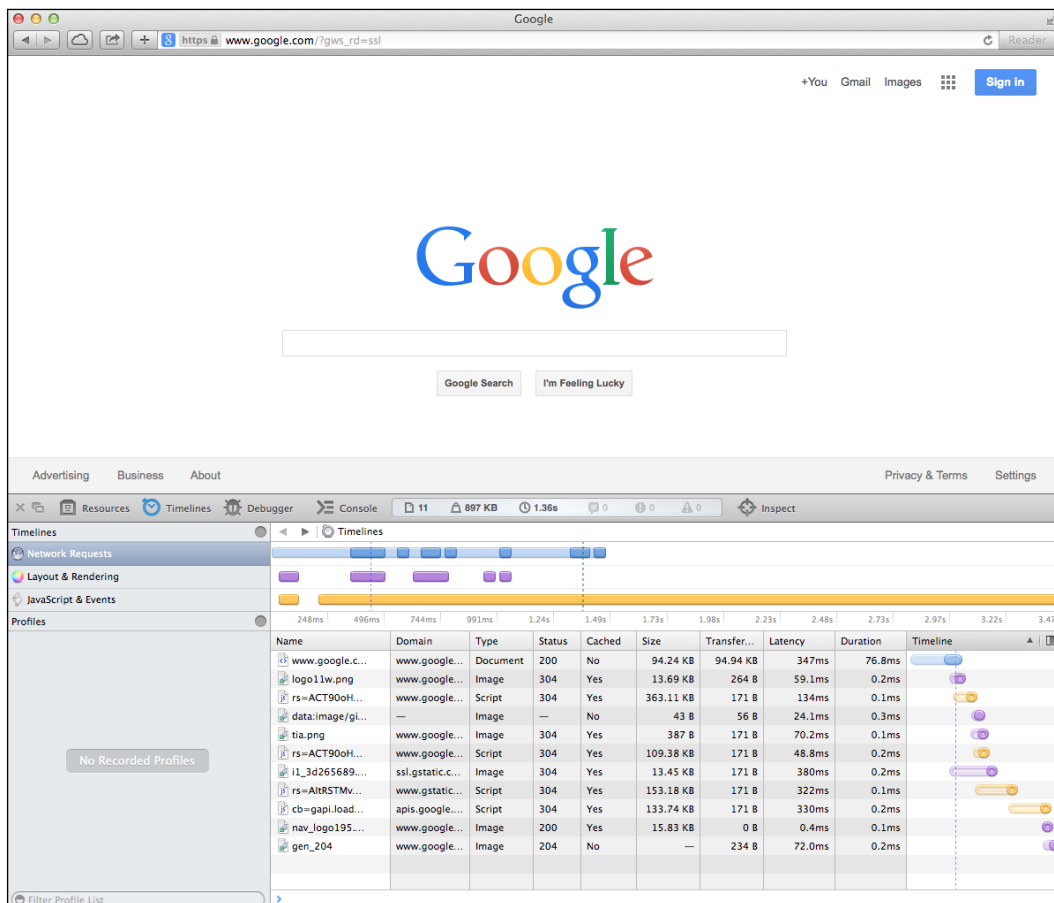
It seems not too long ago that website performance was important but not really required for most web sites. Even during the early days of the Web, it wasn't uncommon to have a really slow website – not because of connection speeds or server locations or which browser was used, no. In many cases, it was because the JavaScript used to render or create functionality for the pages was slow, *very* slow. Mostly, this was done because of a lack of minification tools and debuggers for JavaScript and a lack of knowledge of common JavaScript practices used today.

It was acceptable to the user that the page content was always slow, mainly because most users used a 56 K modem dialing up to their **Internet Service Provider (ISP)**. The screech of beeps alerted the user to the connection process. Then, suddenly, the user was notified on their desktop that a connection had been made and then promptly opened the default web browser, depending on whether it was Internet Explorer 4 on Windows 95 or Netscape Navigator on a NeXTStep machine. The process was the same, as was the 2 minutes and 42 seconds it took to make a sandwich, waiting for HotBot to load.

As time moved on, users experienced Google, and then suddenly, page speed and load times seemed to grab more users' attention, though, even today, the plain Google theme on the main Google search site allows for speedy download of the full site's code. This was regardless of the Internet connection, a whole 1.36 seconds, as indicated by Safari's Timeline tool, shown in the following screenshot, giving us a clear indication of which resources were downloaded the fastest and which ones were the slowest.

Part of the reason for this was that the tools used in modern browsers today didn't exist for Internet Explorer or Netscape Navigator. In the early days of debugging, JavaScript results were debugged using JavaScript alerts, giving feedback to developers since the modern tools weren't around then. Also, developer tool sets today are much more advanced than just simple text editors.

In the following screenshot, we show you a website's download speed using Safari's Web Inspector:



Getting faster

JavaScript, by nature, is a pretty easy language to build. One advantage that JavaScript has is that JavaScript is an interpreted language, which means that the code developed can still be deployed, and even work, according to a project's specifications.

Non-compiling code is both good and bad. Without the need to compile, a developer can quickly build a web page on a full web application in a very short amount of time. Also, it's very approachable for new- or intermediate-level developers in general, making staffing for web projects a bit easier.

Now, what's bad about not using a compiled language is that JavaScript doesn't compile and common errors tend to get missed by the developers involved; even if the code appears to be working, it may not be working efficiently. During the days where developer tools were most likely to be Notepad on Windows and a web browser, any errors were apparent to a user only, leaving out any issues with regard to code performance.

Today, we have various tool sets and build systems on top of our JavaScript skills. It's important to understand that having deep JavaScript knowledge can help you write and review better JavaScript code but, in many cases, we as developers are only human, and we make common mistakes that affect our JavaScript code — not adding spaces after a function's starting brackets or forgetting a semicolon at the end of our code statements, for example.

Choosing a proper editor for a given project that includes basic error-checking before writing a single line of JavaScript can improve the performance and quality of our codebase dramatically, without learning anything new in terms of the inner workings of JavaScript.

Selecting an effective editor

Picking a good editor can greatly affect your code quality as well as your productivity in terms of how fast a project can be coded. As noted in the preceding section, we developers are human, we make mistakes, and it's easy for us to write bad JavaScript, no matter what the skill level of the developer is. So, it's important for us to know when it is appropriate to use one editor over the other. To cover this, I will be breaking up different JavaScript code editors into one of four categories as follows:

- Integrated Development Environments
- Mid-range editors
- Lightweight editors
- Cloud-based editors

Each type of editor has its own strengths and weaknesses, and we will review when to use one over the other, starting with the biggest. The intent is to show when it's appropriate to move from a larger code editor to a smaller editor in terms of JavaScript development.

Integrated Development Environments

Integrated Development Environments (IDEs) are very high-end software tools that not only provide code editing, but also code-organization tools, built-in testing tools, code-optimization scripts, source-control integration, and usually deep code hinting and completion support.

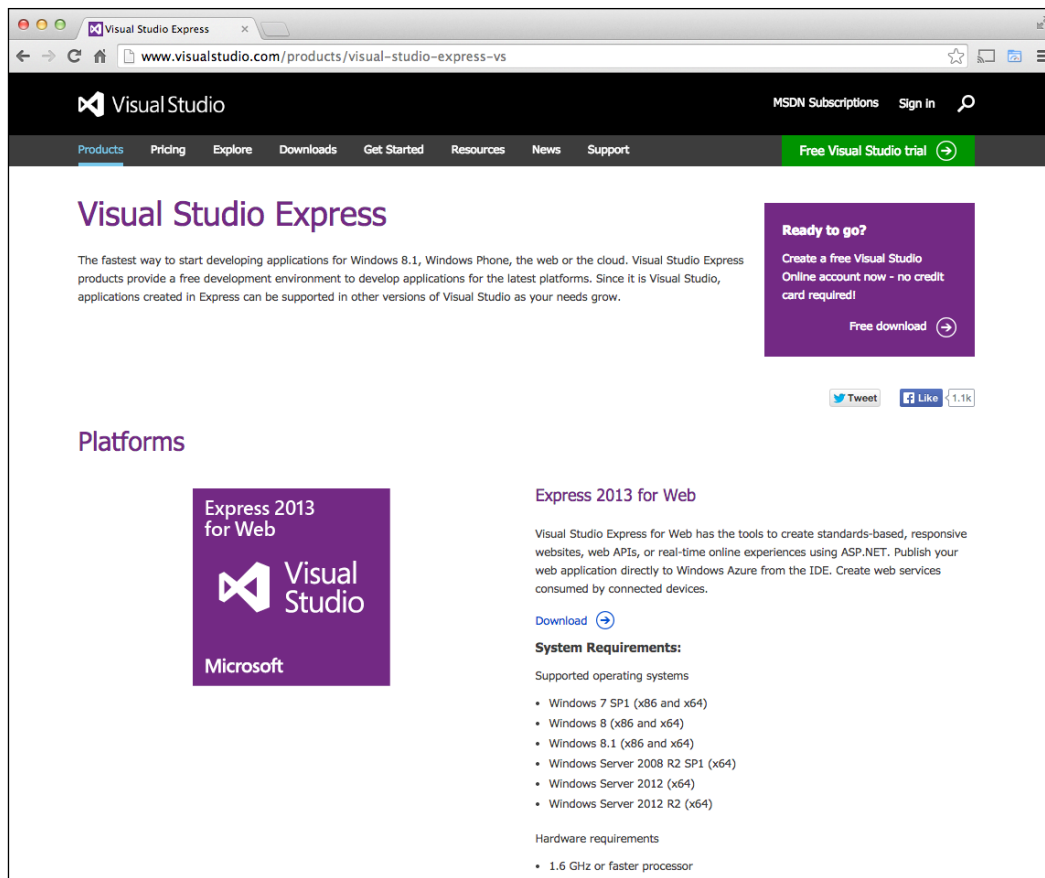
The downside of using an IDE is that the IDE is designed to constantly check the code as the file is being updated while code is being written. This causes the editor to be sluggish and unresponsive at times and painful to use on slower systems. Typically, JavaScript developers tend to dislike the sluggishness of these IDEs and move on to other faster editors.

This can cause issues when large projects kick off, and users use an editor that is ill-suited to structure JavaScript in a proper manner. It's usually recommended that you start with an IDE and work down when a project only requires minor tweaks.

Some popular IDEs for JavaScript are discussed in the upcoming sections.

The Microsoft Visual Studio IDE

If any software is directly associated with the term "IDE", Visual Studio is one. Microsoft Visual Studio IDE can be seen in the following screenshot:

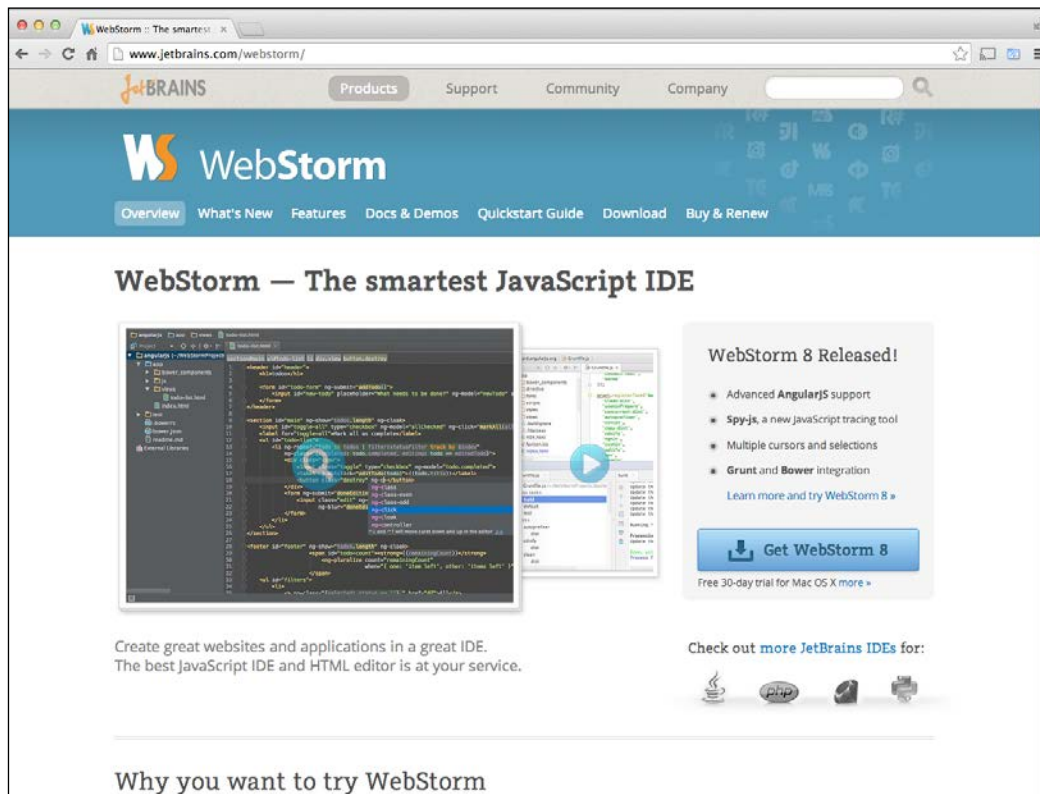


It handles multiple languages, including HTML, CSS, and JavaScript, while handling other language such as C#, Visual Basic, Python, and so forth. In terms of JavaScript, Visual Studio will check deeply within a project's JavaScript code flow and look for minor errors that many lighter editors won't find.

For JavaScript developers, the Visual Studio Express Edition for Web should be powerful enough for any JavaScript projects.

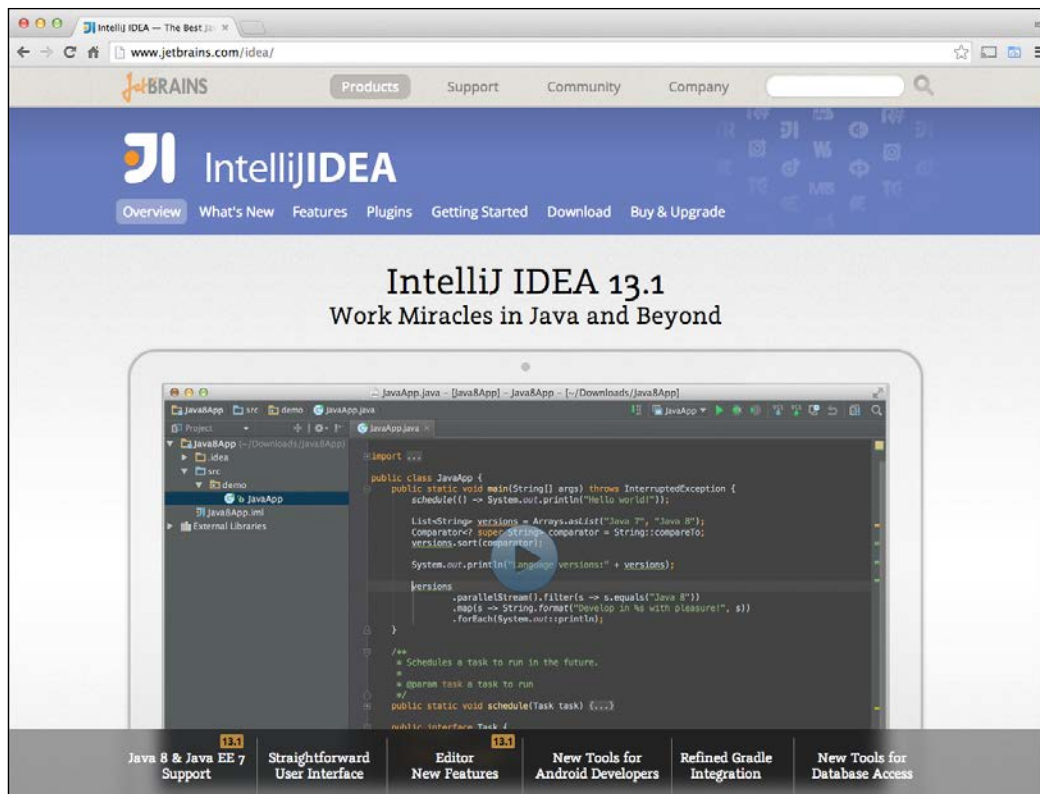
JetBrain's WebStorm IDE

For the JavaScript developer not fond of ASP.NET and looking for a dedicated JavaScript IDE, and/or requiring a Mac or Linux solution, look any further than JetBrains's WebStorm IDE shown in the following screenshot:



This IDE is targeted at JavaScript development, and it handles *any* JavaScript technology you can throw at it: node, AngularJS, jQuery... the list goes on and on with WebStorm. It also has full code hinting and error checking support, similar to Visual Studio, and it has very strong source control support, including Git, SVN, and even Microsoft's Team Foundation Server.

Now sidebar on JetBrains, WebStorm is a lower-tier IDE when compared to IntelliJ IDEA, which is JetBrains's flagship editor for *every* language. The user interface of the IntelliJ IDEA editor can be seen in the following screenshot:



Typically, IDEA is known best as a Java-focused IDE, but it includes the same tools as WebStorm plus many more. Like Visual Studio, it can handle multiple languages, but that comes at the cost of performance. For example, if we started working in both environments on a slower system, we might notice more lag on IDEA than WebStorm when working day-to-day on JavaScript projects.

Again, this is due to the large number of features the IDEs require to be running in the background to make our code better, which is more marked on IDEA; so, again, starting off in an IDE is great to build a well-structured code base early on, but as time progresses and we work repeatedly in a slow editor, we will need something faster with a good base already set up.

With that in mind, many developers who don't see performance issues with an IDE tend to stick with the IDE they've chosen; other developers, however, move on to editors such as the ones in the next section.

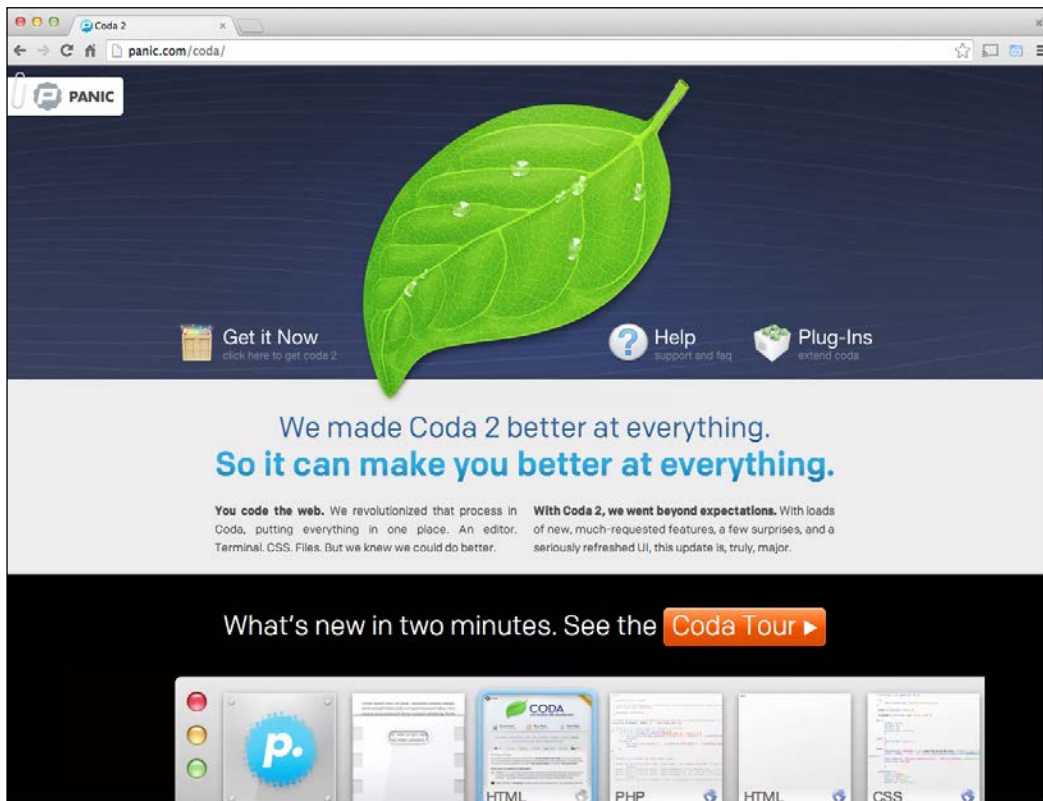
Mid-range editors

Mid-range editors work very well with projects already past the early phases of development or projects that are very small. An exception to using an IDE early on is small projects. These are typically content-management system-based sites, such as WordPress, Joomla, Drupal, and so on, where most of the JavaScript is written for the developer and tested already.

They are also useful for light code hinting, and some can connect to either a source repository or an FTP to push code up. The real differences between these and an IDE are the speed of the editor and the lack of code quality features. Many of these editors only look for glaringly obvious errors in the code, such as missing a semi colon in JavaScript. Nevertheless, they are very useful all-round editors.

Panic's Coda editor

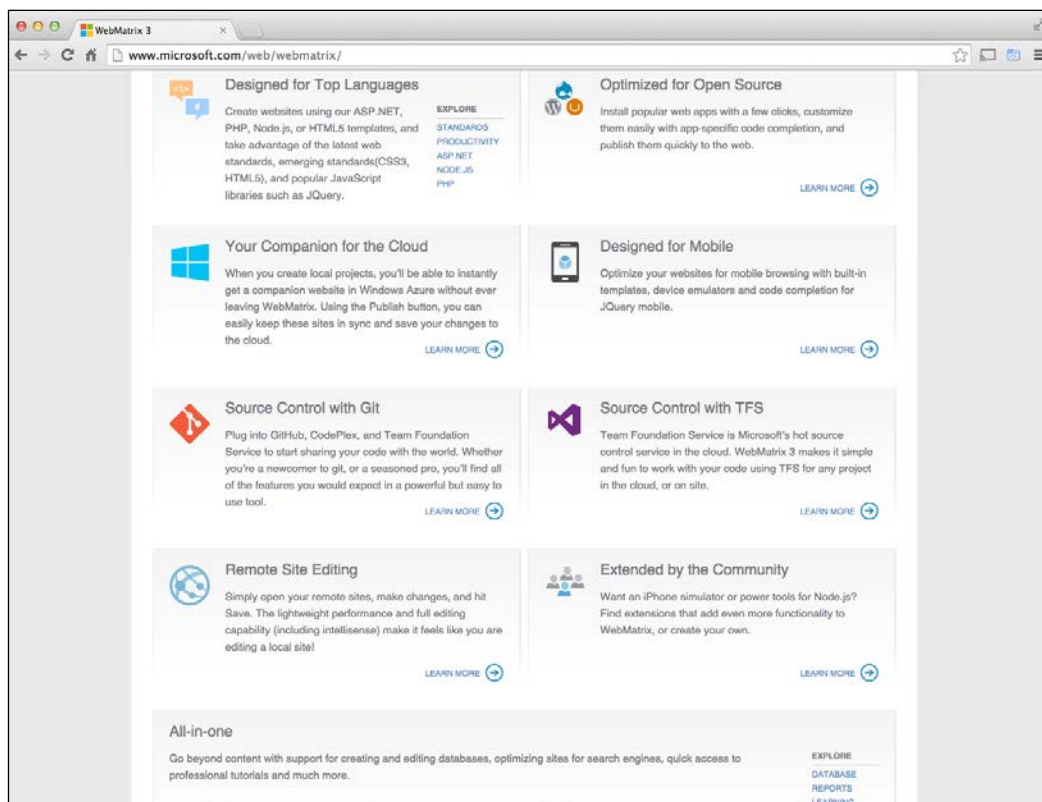
Coda is a Mac-only editor, but it supports HTML, CSS, and JavaScript coding. The following screenshot shows you the user interface of Coda:



It also has some support for Python and PHP, but it's not dedicated to running non-web code on its own. It also features a manual validation checker of JavaScript rather than one that's continuous so, again, there's some support to improve your JavaScript and web code, but it does not always check for errors fully while you code.

The Microsoft WebMatrix editor

WebMatrix is Microsoft's lighter website editor in this mid-range category. It has Git and Team Foundation Server support as well as support for ASP.NET projects, PHP, and NodeJS. The user interface of WebMatrix can be seen in the following screenshot:



WebMatrix is an example of a mid-range editor where you may want to consider an editor's features when choosing which editor you want to use for your project.

For example, if you needed Mac support with Python, then Coda is a good fit, while WebMatrix gives a different set of features, including ASP.NET support. This is a common theme in mid-range editors, where many of them are really designed to do certain things and give just about the minimum support for a code base while keeping the editor as speedy as possible.

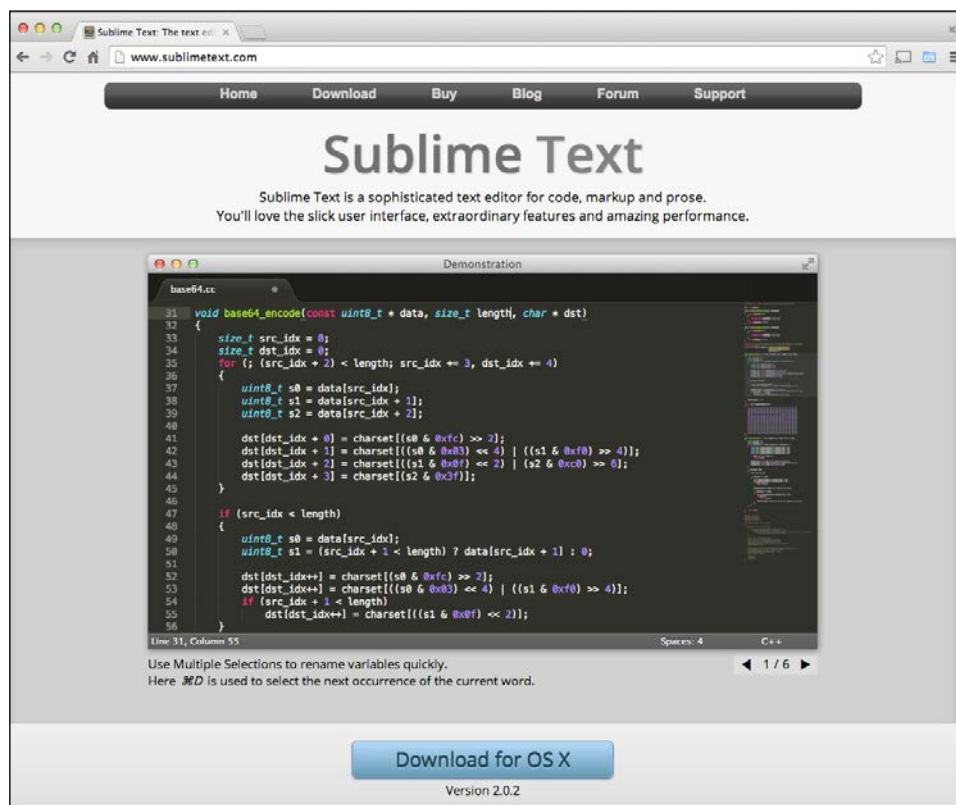
With any editors of this type, we can see that they allow us to connect to an existing project easily and perform some code-checking while working on a fairly fast editor.

Lightweight editors

There are times where we as JavaScript developers *just don't care* about the backend platform a project is using and only need a simple text editor to write a bit of JavaScript code or update an HTML file. This is where lightweight editors come in.

The Sublime Text editor

Sublime Text is a very popular, cross-platform, lightweight editor. Its user interface can be seen in the following screenshot:

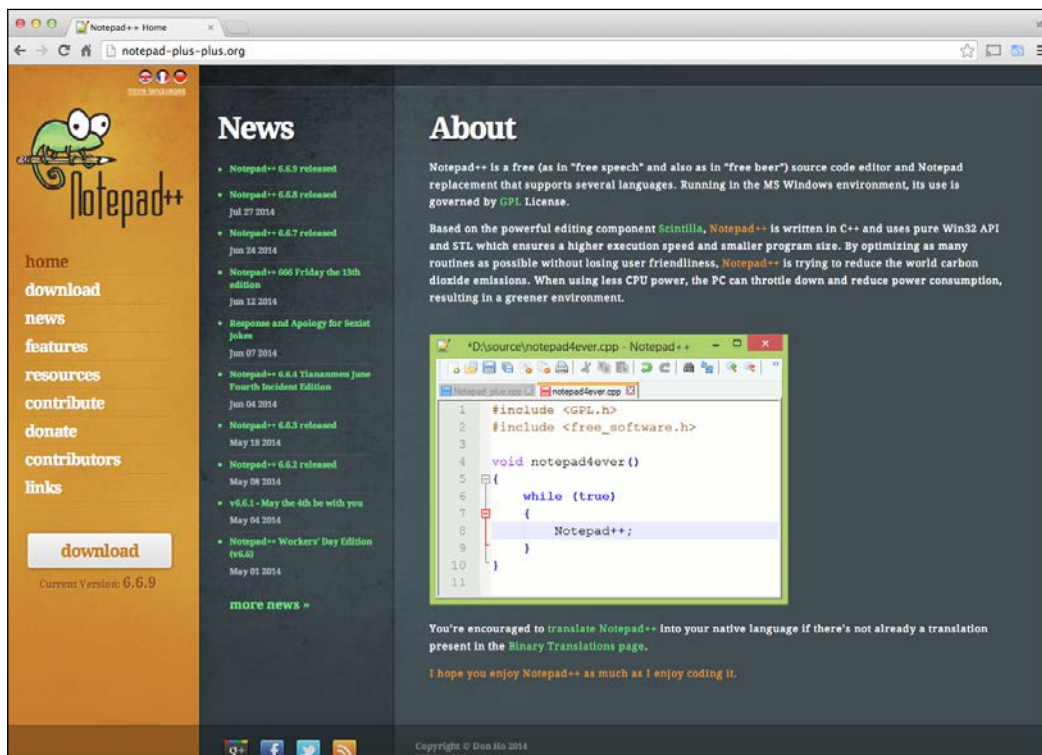


It is well known for its start-up and usage speeds as well as some basic editing features, such as language color hinting and basic code-hinting with multiple language support.

It also has its own package manager called **Package Control**, which allows you to augment Sublime Text to automate some common code-editing and compiling processes. Freshly downloaded, though, it's extremely lightweight and allows developers to add in common plugins required for their development workflow.

The Notepad++ editor

The user interface of the Notepad++ editor is shown in the following screenshot:



On Windows, a JavaScript editor that is exclusive to the Windows platform and that's actually an editor and not an IDE is Notepad++. Similar to Sublime Text, Notepad++ is mostly used as a text editor and has plugin support but doesn't use a package manager such as Sublime Text, so the application runs extremely fast even with plugin support. It also has code-hinting support for some project files, including JavaScript.

In the case of either of these editors, or any other lightweight editor, as they typically don't have code validation included, they make code updates easily and quickly with validation running in the background, at the risk of writing slow or broken code.

Cloud-based editors

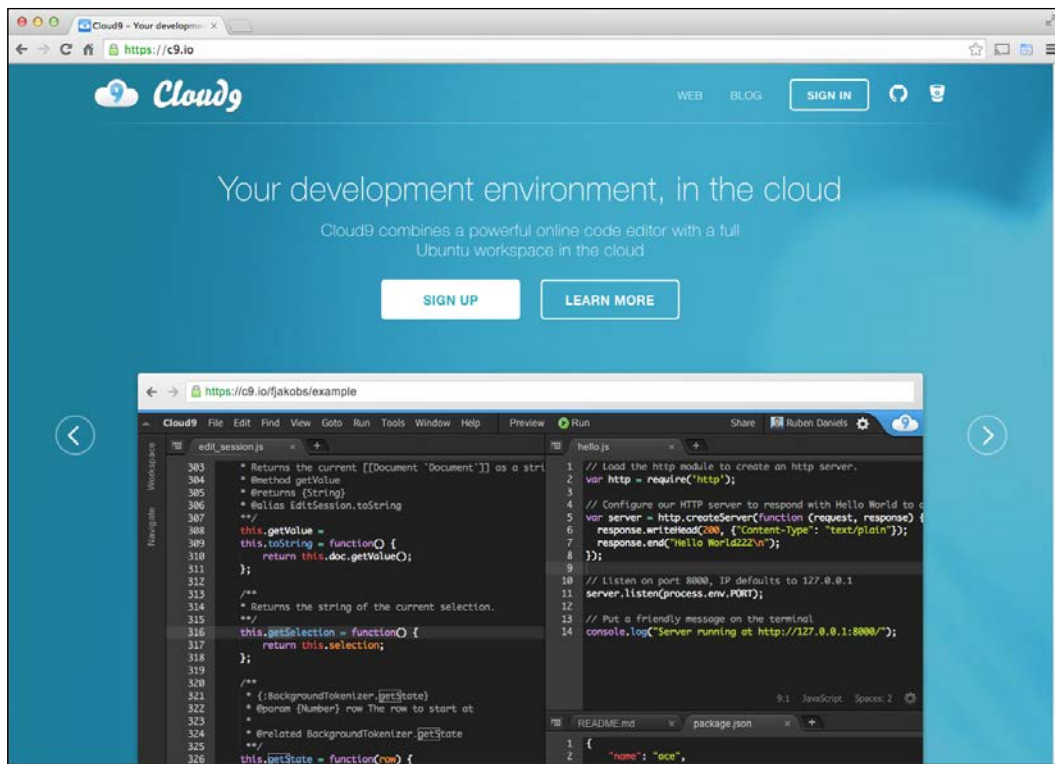
Lastly, cloud- or web-based editors are the shiny new tools new to the web developer's tool belt. They allow a developer to work on a code base inside a browser either as a plugin to a web browser or purely online, and it allows a developer to work on any OS platform, Chrome OS, iPad, or Android operating systems that you might not consider writing JavaScript in!

The advantage of writing code in a browser is that the project code is hosted online, either in Git or simply in the editor's hosted service. Some plugin editors allow you to work from your computer's hard drive like any other editor but are written in HTML and JavaScript with a backend (such as Python, PHP, or ASP.NET) like any other website.

Typically, these editors fit inside the mid-range editor space in terms of features. However, some of them can offer very little in terms of features beyond being online without installing an editor to a computer, which is why they fall in this category. The upcoming sections give a few examples of popular cloud editors.

The Cloud9 editor

Cloud9 editor, available at <http://c9.io/>, is a general web application IDE but is a cloud app with HTML5, PHP, Node.js, Rails, Python/Django, and WordPress support. The following screenshot displays the user interface of the Cloud9 editor:

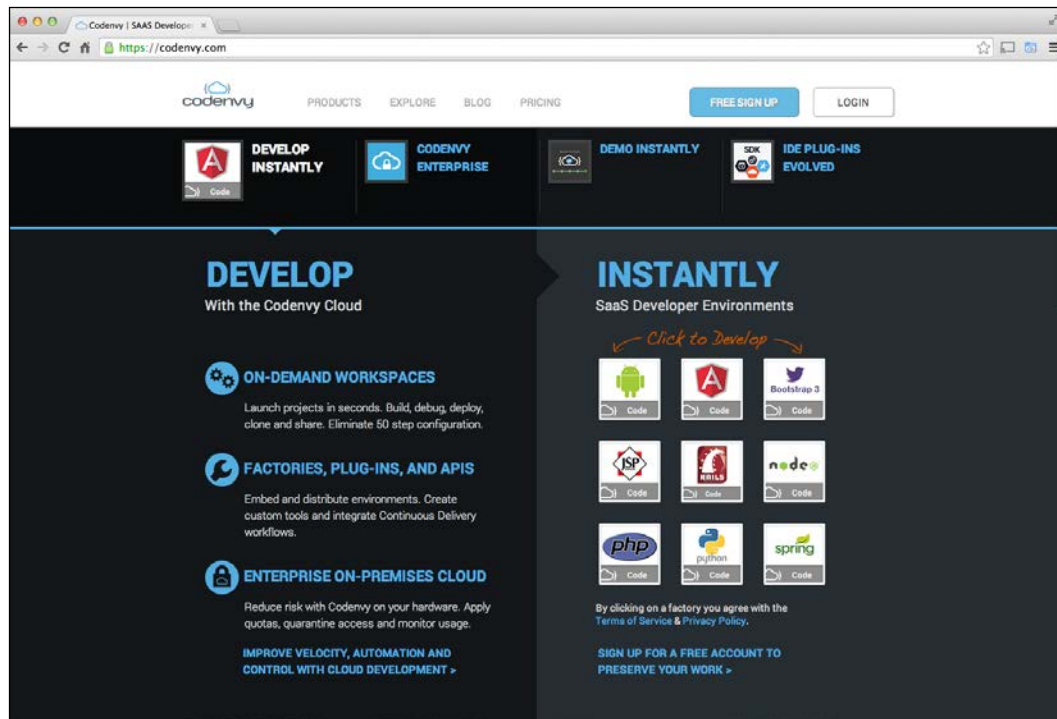


It also allows you to clone from a Git URL or from a GitHub project, so you can choose to have your code hosted in Cloud9 or synced to your own Git repository.

Another feature of Cloud9 is virtual-machine support from the browser for iOS simulator testing as well as console support for Node.js — again, in a browser.

The Codenvy editor

Another online IDE – Codenvy – is available at <http://codenvy.com/>. Its user interface can be seen in the following screenshot:



This editor is pretty similar to Cloud9, but it hosts cloud service projects, such as Google's App Engine. It can also build apps for Android while having full JavaScript support for popular libraries in AngularJS or jQuery.

An issue with cloud editors is that, when JavaScript libraries are involved in a project, an online editor may not be able to recognize library-specific JavaScript or HTML tag conventions used, so it's important to consider features when selecting a cloud editor.

For cloud editors, you can see that they follow a mid-range editor feature-set but allow for quick connection and updates for existing projects.

Summary

In this chapter, we looked at the history of JavaScript's performance and learned how it became a focus for developers and businesses. We also reviewed the four types of JavaScript code editors, and we now understand how to move away from large IDEs for brand new projects, working down to lightweight editors for small updates and changes.

In the next chapter, we will look at how we can keep our code's performance quality high when using a lightweight editor.

2

Increasing Code Performance with JSLint

In this chapter, we will learn about confirming performance fixes in JavaScript, and we will also learn about JSLint. There are two very good JavaScript validation and optimization tools, and we will learn how to use both and how to set the options to get the best code performance optimization results.

So, we are going to cover the following topics in this chapter:

- Checking the JavaScript code performance
- What is JavaScript linting?
- Using JSLint

Checking the JavaScript code performance

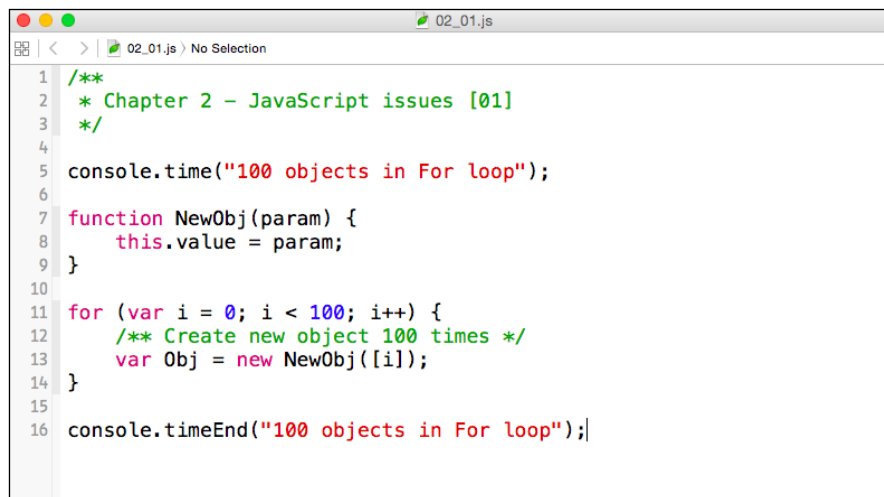
Before we can start talking about how to improve our JavaScript performance, we have to ask ourselves a hard question about what code improvements actually improve the JavaScript application speed. In the earlier days of JavaScript development, many performance improvements were mainly implemented based on known JavaScript coding standards, watching for global variables without the variable being declared, keeping the variable scope in line, and so on without much verification beyond anything visual inside a website.

Today, we have new APIs to take advantage of this problem and scope solutions for small parts of our code.

About the console time API

To solve this issue, modern browsers implemented new console functions called `console.time` and `console.timeEnd`. What these two functions do is allow a developer to specify a label for the `console.time` and `console.timeEnd` functions, measure the amount of time a code block between the `time` and `timeEnd` instances would need to function, and finally, show the result in a console.

Let's take a look at how to use `console.time()` and `console.timeEnd()` in a working example. Here, in our `02_01.js` example file, we have a simple code block creating 100 simple JavaScript objects using the `new` keyword inside a `for` loop, as shown in the following screenshot:



```
1  /**
2   * Chapter 2 - JavaScript issues [01]
3   */
4
5  console.time("100 objects in For loop");
6
7  function NewObj(param) {
8      this.value = param;
9  }
10
11  for (var i = 0; i < 100; i++) {
12      /** Create new object 100 times */
13      var Obj = new NewObj([i]);
14  }
15
16  console.timeEnd("100 objects in For loop");
```

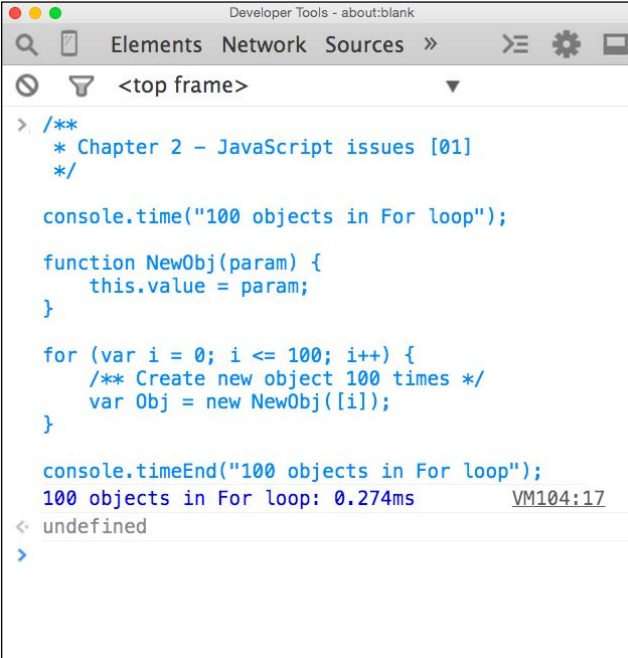
Downloading the example code



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.

As we can see on line 5, we call the `console.time` function and, inside as its parameters, we have the `100 objects in For Loop` string label for our code block. We then add a `NewObj` object constructor on line 7. Following that, we have a simple JavaScript `for` loop on line 11 that creates 100 instances of the `NewObj` constructor, passing in the value from each instance from the `for` loop on line 13. Finally, on line 16, we end the time block with the `console.timeEnd` function, using the same label we declared at the start of the `time` instance.

Let's try this code out in a browser; I'll be using Google Chrome for this, but any modern browser, such as the latest version of Firefox, Internet Explorer, or Safari, should be fine. We'll open `about:blank` in our browser's URL so that we can have a simple test environment to work in, and then we will open our Web Inspectors or browser debuggers, paste the code snippet into our console, and press *Enter*. Here are the results that are displayed in my browser:



```
> /**
 * Chapter 2 - JavaScript issues [01]
 */

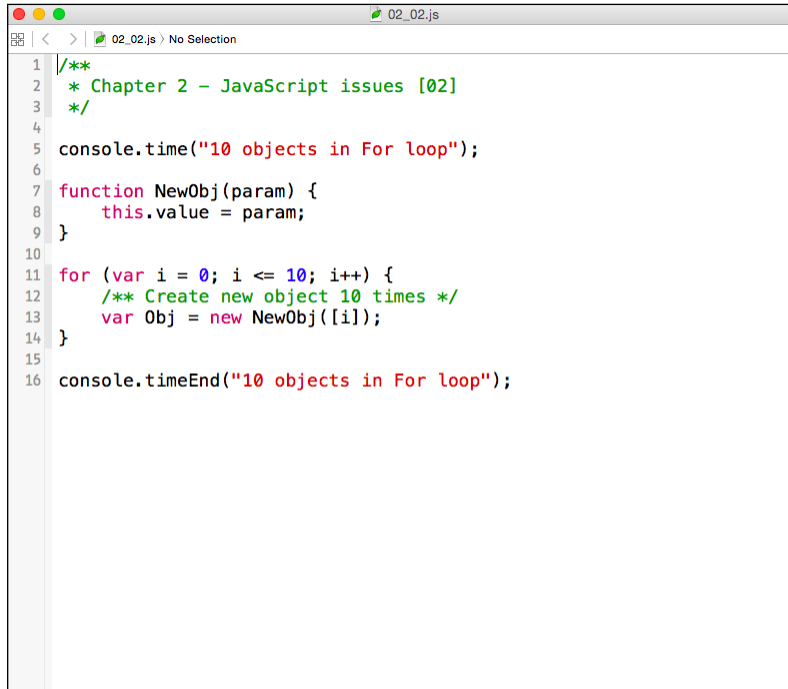
console.time("100 objects in For loop");

function NewObj(param) {
  this.value = param;
}

for (var i = 0; i <= 100; i++) {
  /** Create new object 100 times */
  var Obj = new NewObj([i]);
}

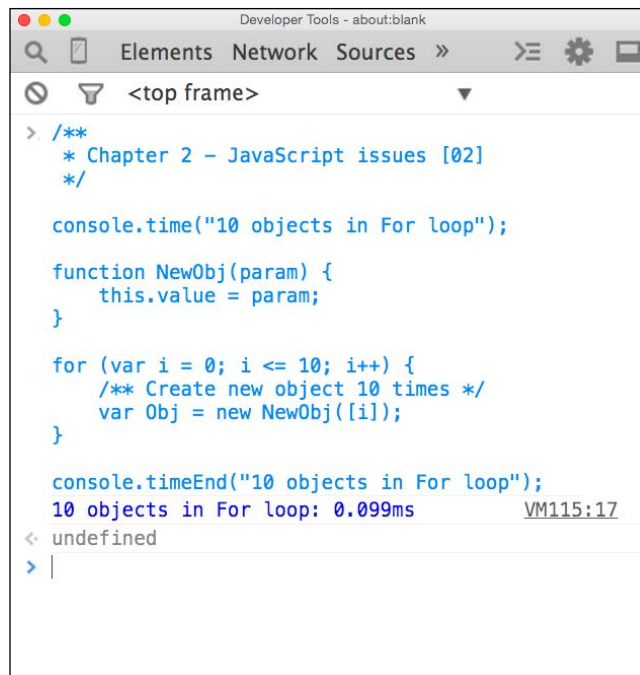
console.timeEnd("100 objects in For loop");
100 objects in For loop: 0.274ms VM104:17
< undefined
>
```

If we take a look just before the undefined line in the output, we can see the label defined in the `console.time` function output: 100 objects in For loop: 0.274ms. So, with some hard data, we can visually see that the block of code we wrote takes the 0.274ms JavaScript interpreter to process that bit of code. Great, but what happens if we tweak our code and make it more efficient, say, by changing our `for` loop to stop at 10 rather than 100. Well, here's an updated code sample for the `02_02.js` file in our example files:

A screenshot of a code editor window titled '02_02.js'. The editor shows 16 lines of JavaScript code. Line 1 is a comment '1 /**'. Line 2 is a comment '2 * Chapter 2 - JavaScript issues [02]'. Line 3 is a comment '3 */'. Line 4 is empty. Line 5 is '5 console.time("10 objects in For loop");'. Line 6 is empty. Line 7 is '7 function NewObj(param) {' with a closing brace on line 9. Line 8 is '8 this.value = param;'. Line 9 is '9 }'. Line 10 is empty. Line 11 is '11 for (var i = 0; i <= 10; i++) {' with a closing brace on line 14. Line 12 is '12 /** Create new object 10 times */'. Line 13 is '13 var Obj = new NewObj([i]);'. Line 14 is '14 }'. Line 15 is empty. Line 16 is '16 console.timeEnd("10 objects in For loop");'.

```
1 /**
2  * Chapter 2 - JavaScript issues [02]
3  */
4
5  console.time("10 objects in For loop");
6
7  function NewObj(param) {
8      this.value = param;
9  }
10
11  for (var i = 0; i <= 10; i++) {
12      /** Create new object 10 times */
13      var Obj = new NewObj([i]);
14  }
15
16  console.timeEnd("10 objects in For loop");
```

Here, we changed the amount the `for` loop iterates on lines 5, 11, 12, and 16; let's run this code and see what happens as shown in the following screenshot:



```
Developer Tools - about:blank
Elements Network Sources >>
<top frame>
> /**
   * Chapter 2 - JavaScript issues [02]
   */
   console.time("10 objects in For loop");
   function NewObj(param) {
       this.value = param;
   }
   for (var i = 0; i <= 10; i++) {
       /** Create new object 10 times */
       var Obj = new NewObj([i]);
   }
   console.timeEnd("10 objects in For loop");
   10 objects in For loop: 0.099ms VM115:17
< undefined
> |
```

We can now see that dropping our `for` loop to 10 iterations versus 100 iterations drops our processing time from 0.274ms to 0.099ms. We can visualize scaling this up to much larger applications knowing that this performance-testing API can be very helpful in evaluating performance in our JavaScript code.

When to use `console.time`

The `console.time()` method allows developers to have an understanding of what code affects performance and what doesn't. The `console.time()` method delivers results based on not only what browser you're using, but also what operating system and system hardware you're using. If you ran the preceding code snippets, they should be near the values given in this book but more than likely not the same by a small variation.

So, when using `console.time()`, think of them as a guide but not as a hard result. As we work through the book using the `console.time()` method, there may be some variation between the results listed here and what you experience depending on your work environment. What should be consistent is that you will be seeing performance improvements in general when using the `console.time()` method.

Now with our performance-testing knowledge in hand, we will start learning about common performance bottlenecks in JavaScript but, before we dig deep into these concepts, we are going to look at tools that help with the evaluation process.

What is JavaScript linting?

Before we talk about JSLint, we need to discuss linters in general, what they are, and how they influence JavaScript performance. A lint is, simply put, a code-validation checker. It allows a developer to point to a code file and check for errors or potential issues ranging from spacing issues to pure code errors.

Linters typically receive the contents of a file and build a source tree. In the case of JavaScript, this can be objects such as global variables, functions, prototypes, arrays, and so forth. After the tree is created, analyzers will take parts of the source tree and report anything an analyzer that was written would flag. Lastly, any rule readers or parameters flagged before running the linter will look for any options to ignore and generate a final report.

Common rule readers for JavaScript options would be settings such as checking for EcmaScript 3, allowing white space, allowing the `continue` keyword, allowing non-strict conditionals for the `if` statements, and so on.

About JSLint

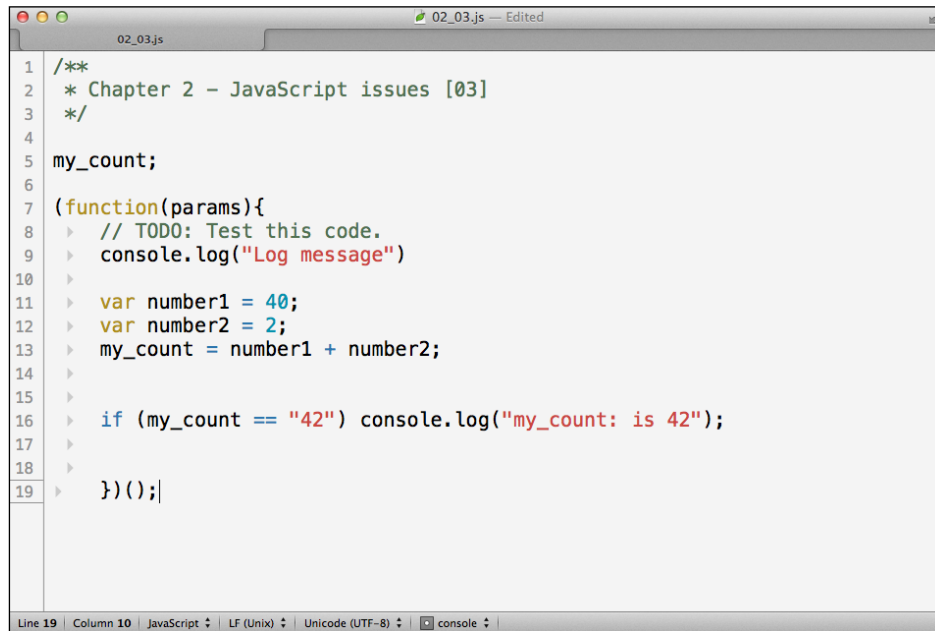
JSLint is a JavaScript code analysis tool that was written in JavaScript by Douglas Crockford, who also helped in popularizing JSON in software development. JSLint can be used in many ways, as mentioned in *Chapter 1, The Need for Speed*. Many IDEs have features beyond just editing code; some of these features include things such as error checking and, in some cases, the IDEs use a version of JSLint.

For this chapter we will discuss how to use JSLint using the official JSLint online site at <http://www.jshint.com/>, as shown in the following screenshot:



Using JSLint

JSLint is very easy to use with a site; all you need is some JavaScript and to paste your code file into JSLint. Let's try a small code sample, as shown in the following screenshot and that you can reference in the example file as the `02_03.js` file:



```
1  /**
2   * Chapter 2 - JavaScript issues [03]
3   */
4
5  my_count;
6
7  (function(params){
8    > // TODO: Test this code.
9    > console.log("Log message")
10   >
11   > var number1 = 40;
12   > var number2 = 2;
13   > my_count = number1 + number2;
14   >
15   >
16   > if (my_count == "42") console.log("my_count: is 42");
17   >
18   >
19  > })();|
```

Line 19 | Column 10 | JavaScript | LF (Unix) | Unicode (UTF-8) | console

Now, let's paste our code into the input box present at <http://www.JSLint.com> and click on the **JSLint** button. Immediately, we should see a list of errors showing up on the site below the **JSLint** button, as shown in the following screenshot:



See also these remaining errors:



Reviewing errors

Before looking at these errors, let's look at the bottom of the error list; we will see an error: `Stopping. (52% scanned).` This is a warning that JSLint found so many errors that the analyzer tools in JSLint simply gave up reviewing errors. It's important to keep this error in mind when reviewing JSLint messages; as only 52 percent of the code was reviewed, additional errors may appear once we fix them.

Okay, now that we've understood JSLint's limitation, let's fix these errors. When working with JSLint, work through the error list top down, so error 1 is Unexpected character ' ' (space) ' .. Well, what does that mean? To explain, JSLint is *very* picky about the way spacing should be in JavaScript files. A JavaScript interpreter assumes a specific spacing within certain JavaScript objects and variables.

This empty space is displayed before any other errors in the code appear, so we can assume that this error comes before any code appears and, in fact, that is the case. If we look at the 02_03.js file, actually line 4 is causing the issue, which is the space between the comment header and our my_count global variable.

Configuring messy white space

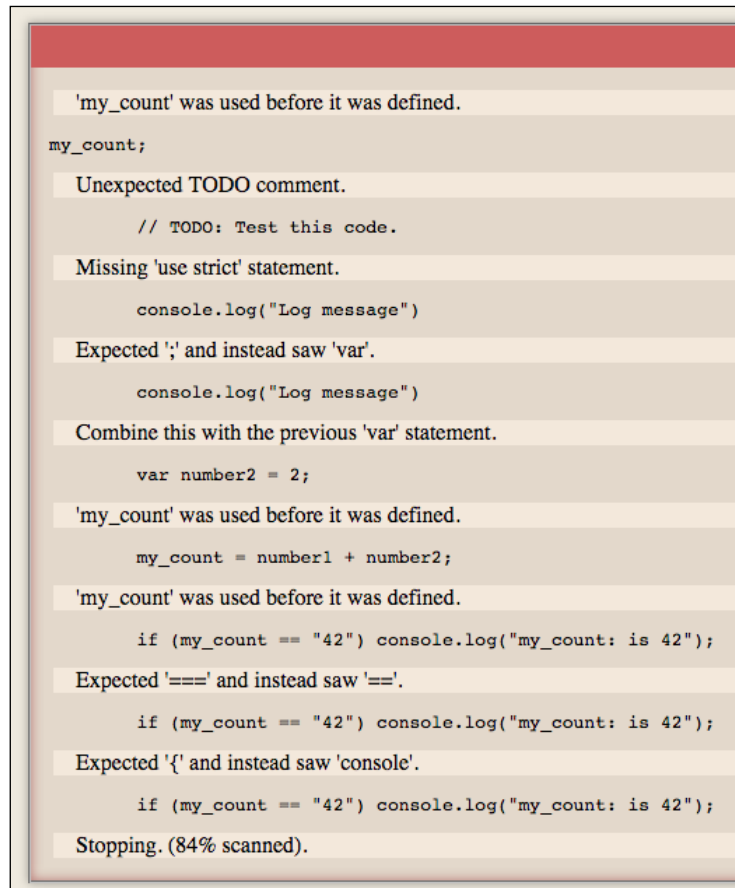
We can address our spacing errors in two ways: we can review each space and correct it line by line or, if we are using a minifier, we can tell JSLint to ignore empty and unnecessary lines. To do this, we will navigate to **Options** on the bottom of the page, and we will set the **messy white space** option to **true**. This will tell JSLint to ignore any spacing issues that aren't directly associated with code interpretation, as indicated in the following screenshot:

The screenshot shows the JSLint Options panel. It has a title bar with 'Options' and a 'clear options' button. The panel is divided into several sections:

- Assume...**: A list of assumptions with checkboxes. 'a browser' is checked. Others include CouchDB, console, alert, ...; Node.js; Rhino; and Stop on first error.
- Tolerate...**: A list of tolerance options with checkboxes. Checked options include assignment expressions, bitwise operators, continue, debugger statements, == and !=, and eval. Other options include unfiltered for in, uncaptialized constructors, dangling _ in identifiers, ++ and --, and [^...] in /RegExp/, and unused parameters.
- Tolerate...**: A list of tolerance options with checkboxes. Checked options include missing 'use strict' pragma, stupidity, inefficient subscripting, TODO comments, many var statements per function, and messy white space (which is set to true). Other options include Indentation, Maximum line length, and Maximum number of errors.
- predefine global variables here**: A text input field.
- JSLint Directive**: A section with a 'select' button and a text input field containing the directive `/*jshint white: true */`.

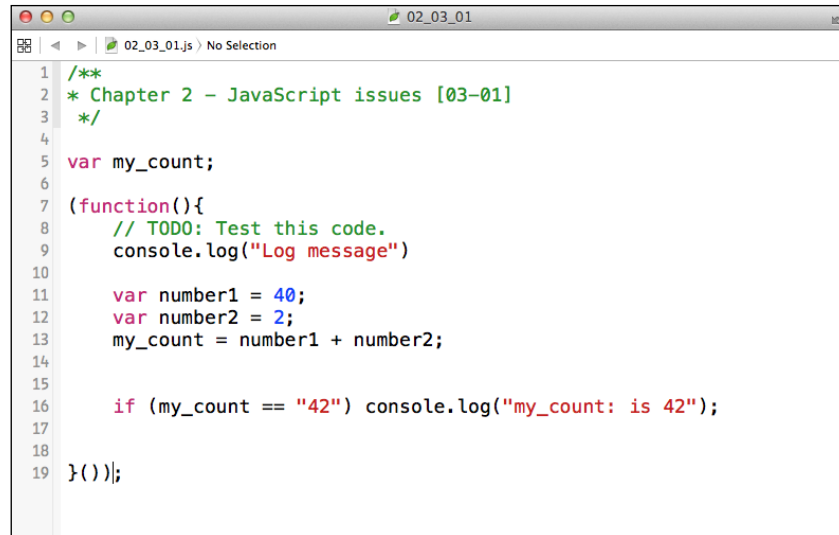
Once this is set to **true**, we will see a new panel appear under options, called **JSLint Directives**. The **JSLint Directives** panel provides a quick list of the parameters JSLint will pass in when reviewing code, before executing the validator. Seeing directives here is very helpful if we're trying to copy-and-paste this configuration in another instance of JSLint, say a build system... but more on this soon.

With messy white space ignored, we can rerun JSLint on our code and see an updated list of errors, shown as follows:



Now, we check how much of the code JSLint detected. This time, if we look at the last error, we can see that JSLint stopped at 84%, which is much better than before, but we can do better here. Let's take a look at the first new error. At the top of the error list, we can see the error stating 'my_count' was used before it was defined. This is line 5, character 1 in the **Errors** panel.

This indicates that we forgot to declare `var` before our `my_count` variable, so let's update it as shown in the following screenshot, adding `var` to `my_count` on line 5, and then let's rerun JSLint. You can reference the update in the exercise files as `02_03_01.js`:

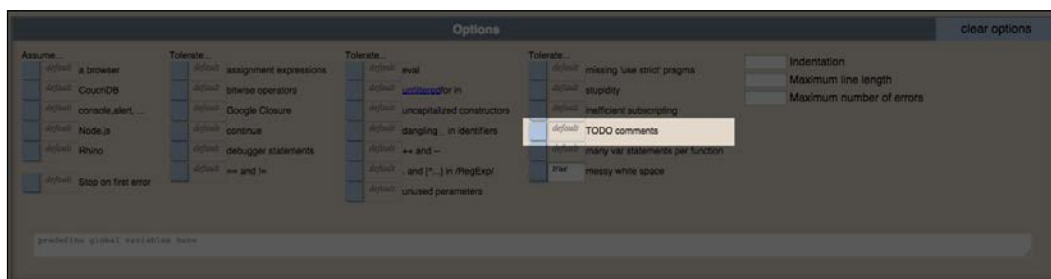


```

1  /**
2  * Chapter 2 - JavaScript issues [03-01]
3  */
4
5  var my_count;
6
7  (function(){
8      // TODO: Test this code.
9      console.log("Log message")
10
11     var number1 = 40;
12     var number2 = 2;
13     my_count = number1 + number2;
14
15
16     if (my_count == "42") console.log("my_count: is 42");
17
18
19 }());

```

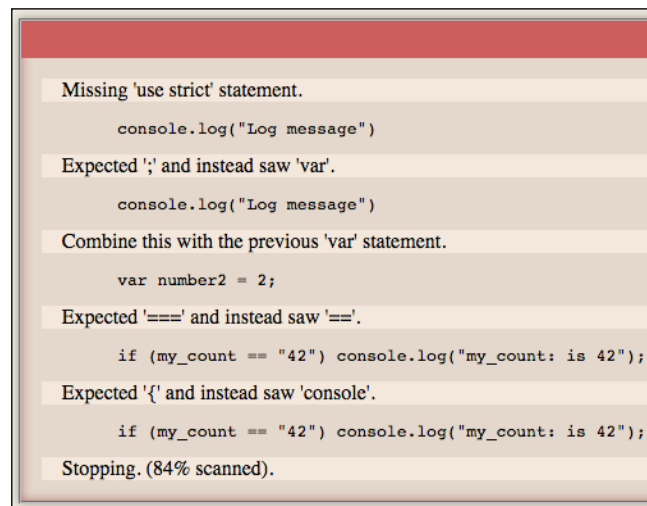
Next, after rerunning JSLint with our updated code, let's look at the next two lines. The first line states `Unexpected TODO comment`. This is pretty straightforward; in JSLint, we can specify allowing **TODO comments** in our JavaScript code, which is pretty handy! Let's allow this as we are just improving our code in JSLint and are not completing the file for now. Take a look at the options I've highlighted, where you can set `TODO` to be allowed or not:



We will now set **TODO comments** in the **Options** panel to **true**; next, let's take a look at the remaining errors.

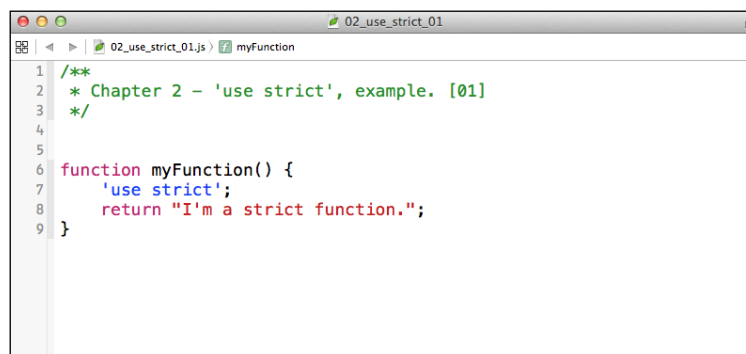
Understanding the use strict statement

So, here's what's now left in our JSLint error list, shown in the following screenshot. The next error we see is Missing 'use strict' statement.. Now, if you haven't seen the `use strict` statement in JavaScript previously, I'll explain:

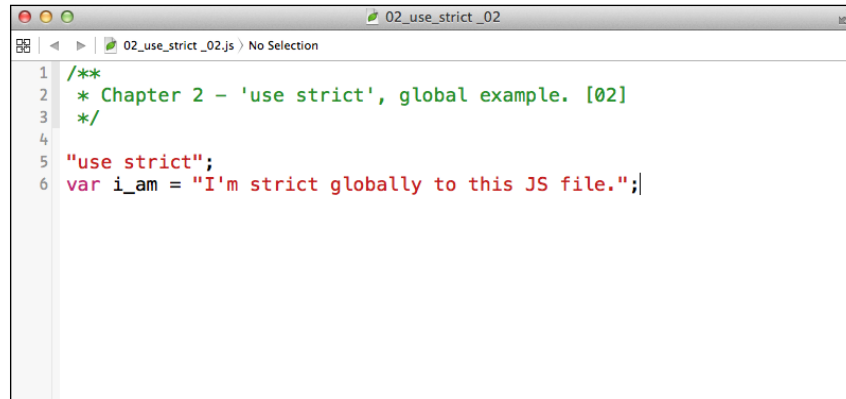


The `use strict` statement is a hint to a browser to enable *strict mode* when reading JavaScript at runtime. This allows errors typically displayed as warnings to return as errors in our browser. One other advantage to using the `use strict` statement in our code is that it allows the JavaScript interpreter to run faster as it assumes the code has been optimized and bug-tested properly. This tells the JavaScript interpreter that code here has been written properly and the interpreter doesn't have to run as many checks on the code at runtime.

Using the `use strict` statement isn't hard to implement, and we can add it before any code inside every function like this:

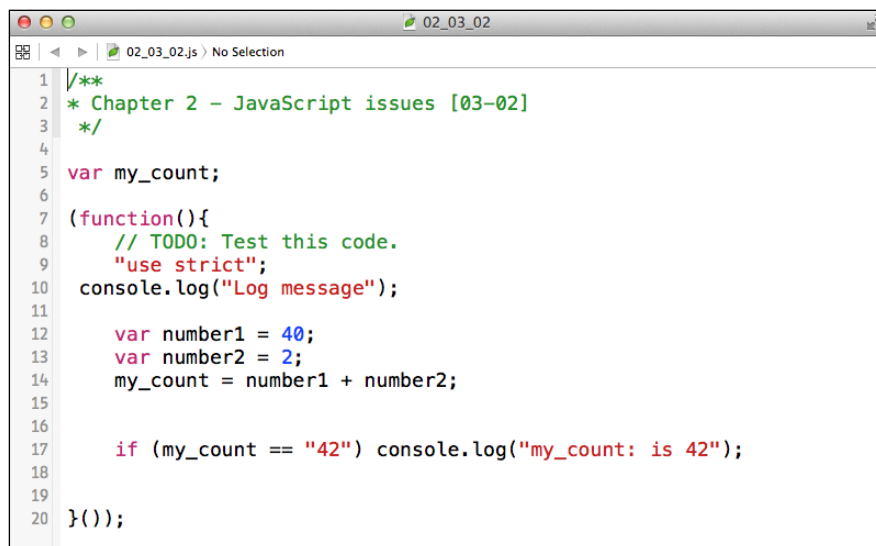


We can also include it globally in the full JavaScript file by adding it above the first line of the code, as shown in the following screenshot:



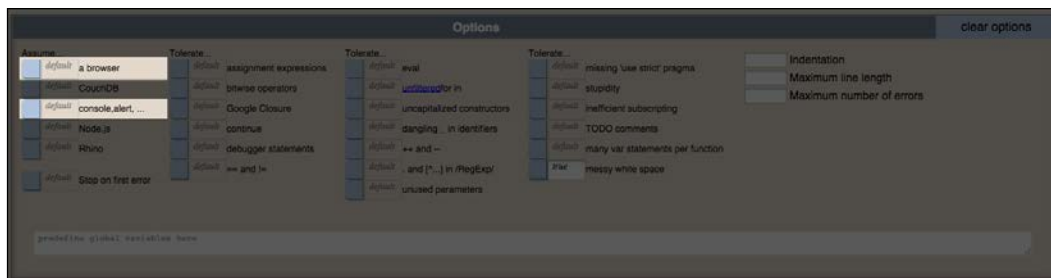
One thing to know about the `use strict` statement with regard to JSLint is that JSLint *prefers* to set the `use strict` statement at the function level (shown previously in the first `use strict` example). The idea is that it properly sets the scope for the `use strict` statement as per the function for better code testing and analyzing, but both are correct in terms of JavaScript.

Let's finish up these remaining issues under our `TODO` comment; on line 9, we will add `"use strict"` followed by adding a semicolon to what would now be line 10 after our `console.log` statement. With that finished, it should resemble the following screenshot:



Using console in JSLint

We are nearly finished with this code. However, on executing it we get a list of errors in which the first line, which may seem odd, states: 'console' was used before it was defined. in the **Errors** panel. JSLint can validate JavaScript that may not be designed for a browser; this could be a Node.js script, for example. To enable browser objects, we need to enable the **console**, **alert**, ... and a **browser** options in our JSLint **Options** panel; we can set these to **true**, as shown in the following screenshot:



With these enabled, lets rerun the script; the remaining errors should be straightforward. The first error complains that we should Combine this with the previous 'var' statement.. We can remove the number1 and number2 variables and simply assign `my_count = 42;`.

Lastly, our `if` statement could use some work. First, JSLint complains that we are using a loose conditional (a double equals for comparison) in our `if` statement. If we use triple equals for comparison, we compare the type as well. By doing this, our code will run the comparison faster than before. Also, the `if` statement doesn't include brackets around the conditional code, and this can slow the interpreter down, so let's add them. Our final code should resemble the following screenshot:

```
02_03_03
1  /**
2   * Chapter 2 - JavaScript issues [03-03]
3   */
4
5  var my_count;
6
7  (function(){
8    // TODO: Test this code.
9    "use strict";
10   console.log("Log message");
11
12   my_count = 42;
13
14   if (my_count === "42") {
15     console.log("my_count: is 42");
16   }
17
18 }());
```

Now let's re-run our finalized code thru JSLint and we should see a screen like this:



We can see that we now have no errors in JSLint, and we can also see a **Function Report** panel, indicating a variable scope as a note of what variables are global to the file and what variables and functions exist inside a function, including our anonymous function example.

Before we wrap up this chapter, let's try the `console.time` method on both our `2_03_01.js` and `02_03_03.js` code files with the `console.time` function wrapped around. The time I get on my end for the former is `0.441ms`, and the optimized code with JSLint is `0.211ms`! Not bad; *double* the performance!

Summary

In this chapter, we learned the basics of the `console.time` and `console.timeEnd` methods, and we also learned about JSLint and how it can improve our JavaScript performance. In the next chapter, we will take a quick look at JSLint and get our hands dirty by integrating it into a build system!

3

Understanding JavaScript Build Systems

In this chapter, we will learn about JavaScript build systems and their advantages for JavaScript performance testing and deployment. We will also incorporate JavaScript code testing into our build system using the knowledge we gained about JSLint in the last chapter.

In short, we are going to cover the following topics in this chapter:

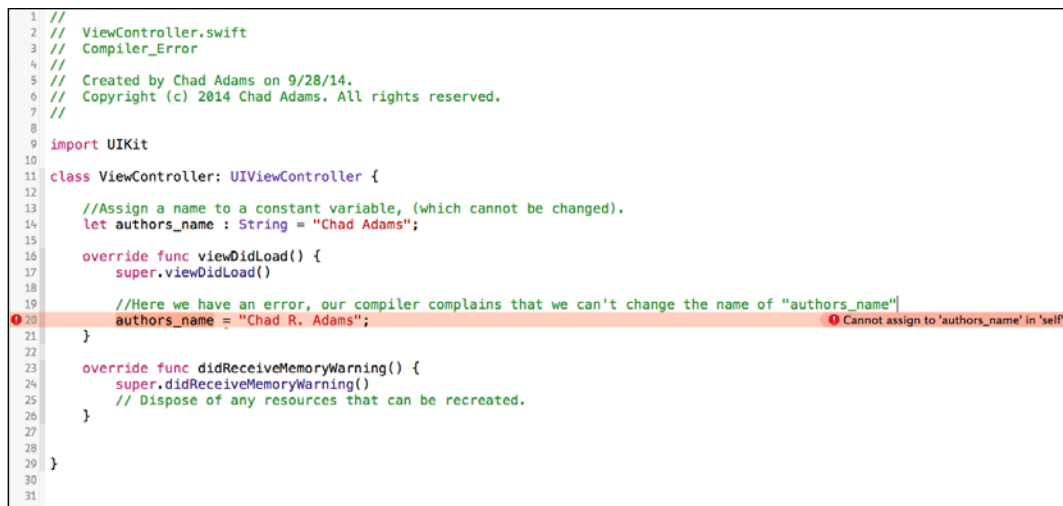
- What is a build system?
- Setting up our build system
- Creating a distribution

What is a build system?

Typically, a **build system** is an automated process that assists developers writing clean optimized code. We may think that such a thing would be standard across all programming languages. Now, compiled languages usually have a compiler; a **compiler** takes a program written by following a language specification, and creates output code compatible with the target machine.

Compiling code by example

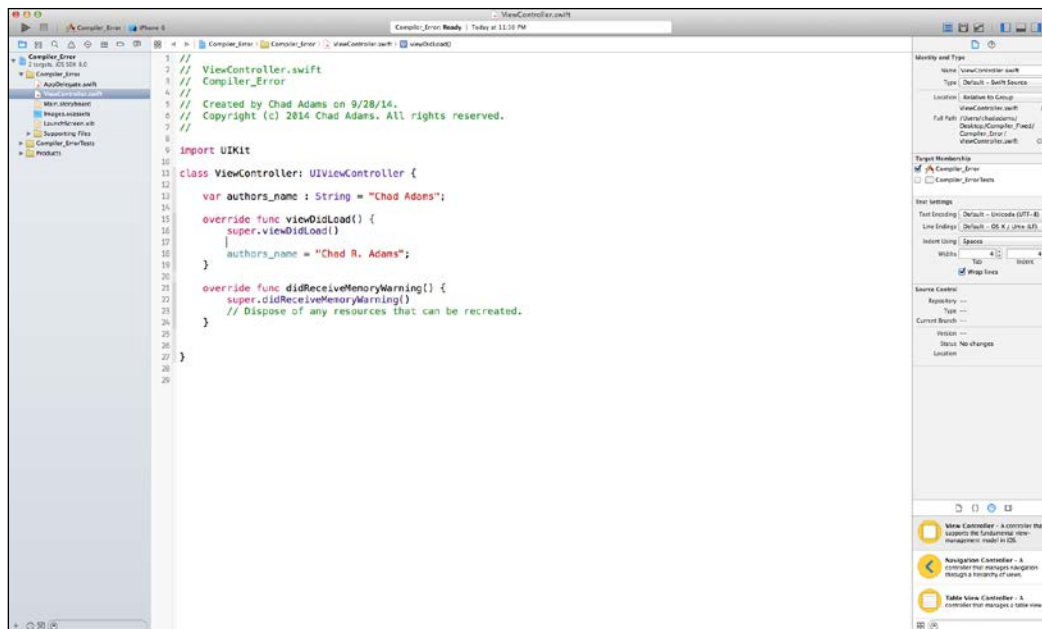
Compilers typically work through a spec when a code file is sent for processing. To keep a compiler from crashing from bad code, a compiler is set up with many error checkers that stop the compiler beforehand and display an alert, thus blocking the compiling process. Now some IDEs allow you to see some of your mistakes before attempting to run your code. The following screenshot shows a simple Xcode Swift file being checked while editing:



```
1 //
2 // ViewController.swift
3 // Compiler_Error
4 //
5 // Created by Chad Adams on 9/28/14.
6 // Copyright (c) 2014 Chad Adams. All rights reserved.
7 //
8
9 import UIKit
10
11 class ViewController: UIViewController {
12
13     //Assign a name to a constant variable, (which cannot be changed).
14     let authors_name : String = "Chad Adams";
15
16     override func viewDidLoad() {
17         super.viewDidLoad()
18
19         //Here we have an error, our compiler complains that we can't change the name of "authors_name"
20         authors_name = "Chad R. Adams";
21     }
22
23     override func didReceiveMemoryWarning() {
24         super.didReceiveMemoryWarning()
25         // Dispose of any resources that can be recreated.
26     }
27
28 }
29
30
31
```

Without getting too technical in iOS development, we can see that, on assigning a constant variable in Swift, if I attempt to change the variable as in the preceding screenshot, my code flags an error.

Now if I change the `let authors_name` constant to a dynamic `var` variable (just like in JavaScript), the error itself corrects, as shown in the following screenshot, and removes the error displayed in the IDE:



Error-checking in a JavaScript build system

In the past, HTML editors for JavaScript and HTML content, such as Dreamweaver, have done this since the creation of early web code editors.

The difference between what's done in Xcode for a compiled language and what's done in a JavaScript IDE is slightly different. With a compiled language, an error must be fixed before a code file can run; this is usually considered as static type checking. JavaScript, however, can run with an error, even when overridden with a try-catch block. Simply put, as stated in *Chapter 2, Increasing Code Performance with JSLint*, JavaScript is an interpreted language, and the *only* language that is really tested for errors at run time.

With that in mind, how do editors such as Dreamweaver, WebStorm, or Visual Studio check for errors then? Well, if you remember in *Chapter 2, Increasing Code Performance with JSLint*, we saw how linting tools provide feedback on potential or verifiable bugs in JavaScript code; this returns a list of errors.

With an IDE, the editor is coded keeping this in mind and takes each error to display it with the associated line and column in the JavaScript file.

So, to make a build system, we will need to incorporate this sort of error checking just like using `http://jshint.com/` but in a more automated fashion. This allows lightweight editors to use the same checking tools that are used in more expensive and heavier IDEs.

Adding optimization beyond coding standards

Like our Xcode example earlier in the chapter, we want our final output to be optimized for our project; to do this, we will add minification to our build system, allowing us to keep a developer version or source project to be saved in a directory with a distribution in another directory. Simply put, minification allows us to compress our JavaScript code causing our web applications to download faster, and run more efficiently.

This can be helpful if we are using source control to maintain our project, allowing us to quickly grab a stable distribution that's optimized, but not easily debuggable, and debug it with our source directory's files.

Now as JavaScript developers, we can even add other minification build options that we may need for the project, such as an image optimizer for our project's image directory, or minify our CSS file and add information comment blocks on top of our JavaScript files. By compressing our JavaScript, the JavaScript interpreter doesn't have to guess the distance in whitespace in our code, which creates more efficient and better-performing code.

Creating a build system from scratch using Gulp.js

Now that we've introduced build systems and the reason for their use, let's go ahead and create a simple build system. Our goal is to create a distribution build from our source directory, a copy that's optimized and ready for production. We will also integrate JSLint, as we learned from the last chapter, to check our code as we create builds for any potential issues that might have been missed during development.

In this chapter, we are going to create a build system to test our JavaScript project. We will also incorporate minification into our build system, and copy files to our build directory. So when we are ready to deploy, our code base is already set to be deployed.

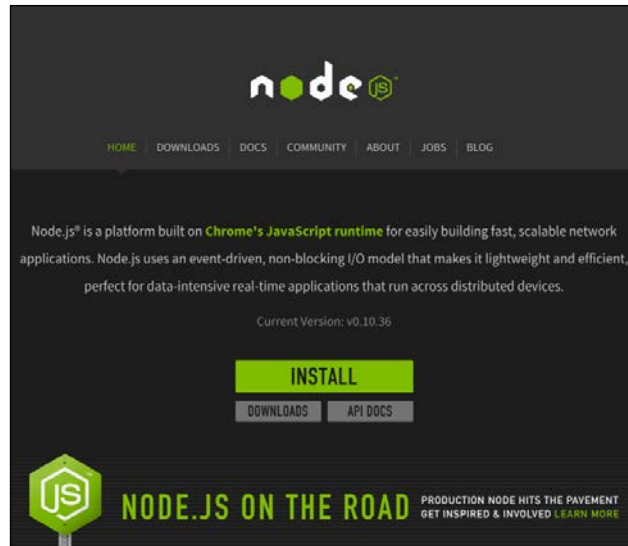
Before setting off on this project, we will need to understand a few technologies specific to JavaScript, particularly the build system that we will want to take into account; we will especially deal with technologies such as Node.js, NPM, Grunt, and Gulp. If you have only heard about these before, or maybe have tinkered with some of these and never really got further than that, don't worry; we will go over each of these one-by-one and understand their advantages and disadvantages.

Node.js

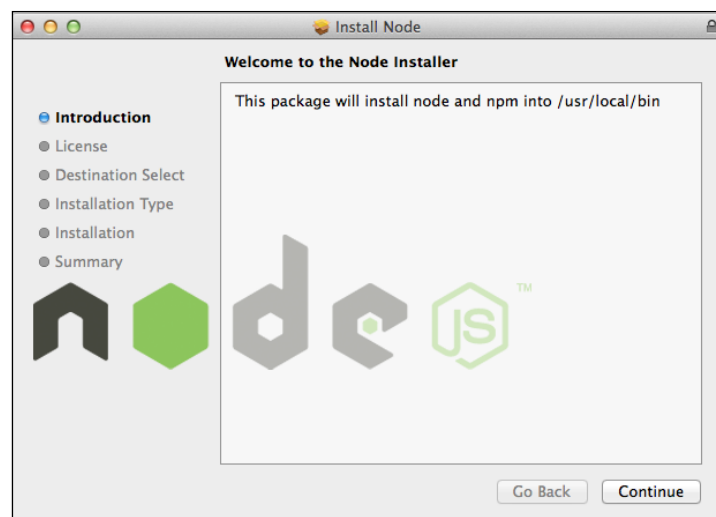
Node.js is a JavaScript interpreter for your operating system. For JavaScript developers, the concept of JavaScript code working as a backend code base like Java or C# may seem odd, but has been shown to work in new creative ways. For example, the community of Node.js developers has created plugins to create custom-built, JavaScript-based applications for the desktop.

This puts JavaScript in a very new place. When traditional application developers complain about JavaScript, one of the main complaints is that JavaScript cannot read or write files to a hard drive, which is usually a very basic feature for a programming language. Node.js allows custom objects to interact with the operating system. These include objects such as `FS` or `FileSystem` that allow for reading and writing files and work pretty much like a console in a web browser.

For this project, we won't discuss Node.js in depth (that's another book), but we will be installing Node.js into our OS so that we can run and test our build system. So let's download Node.js and get started. First, navigate to <http://nodejs.org/> and click the green **INSTALL** button, as shown in the following screenshot:

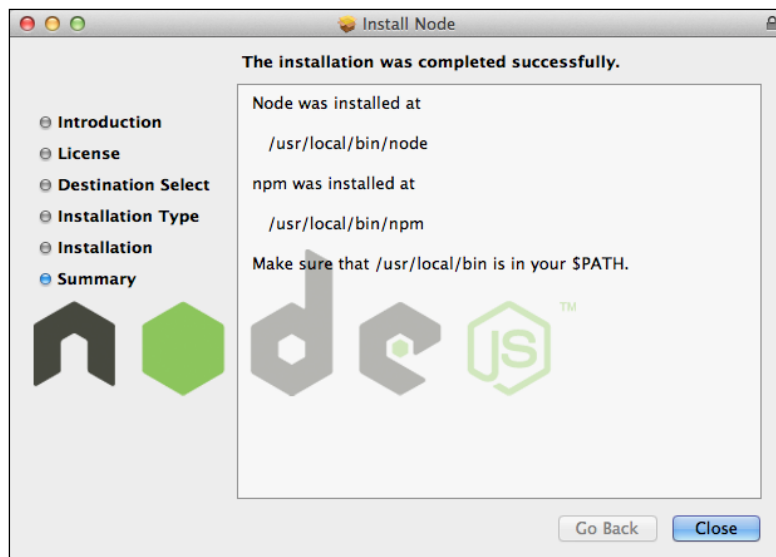


Node.js is cross-platform, and so most of these instructions should work for you. I'll be using a Mac with OS X for this installation's introduction. For most platforms, Node.js will come with either a .pkg or .exe install wizard, as shown in the following screenshot:



From here, follow the wizard, accept the user licenses and install for all users. By installing for all users, we allow Node.js to have full system access which we do want, since some plugins for Node.js may require certain features that are not accessible by a single user or non-administrator.

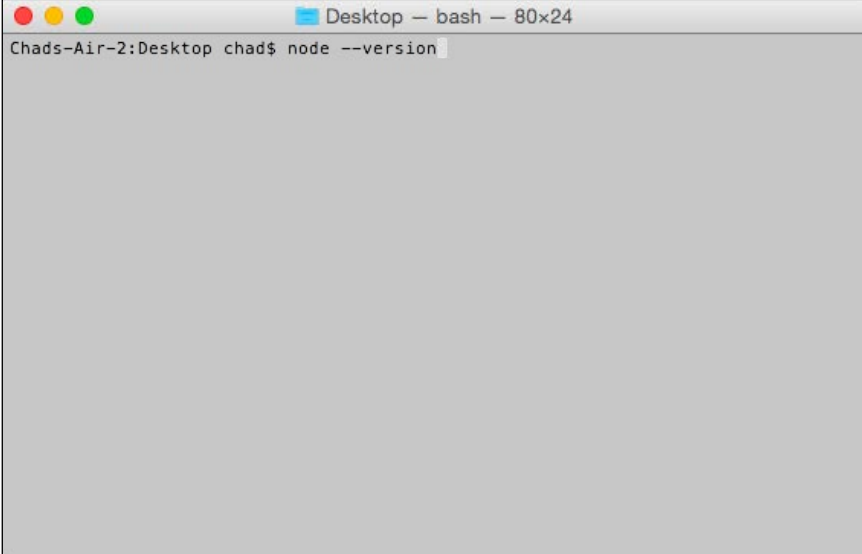
When you've finished installing Node.js, keep in mind the paths set by the installer; if you want to remove Node.js in future, check out the following screenshot to see where the installer has added Node.js:



Testing a Node.js installation

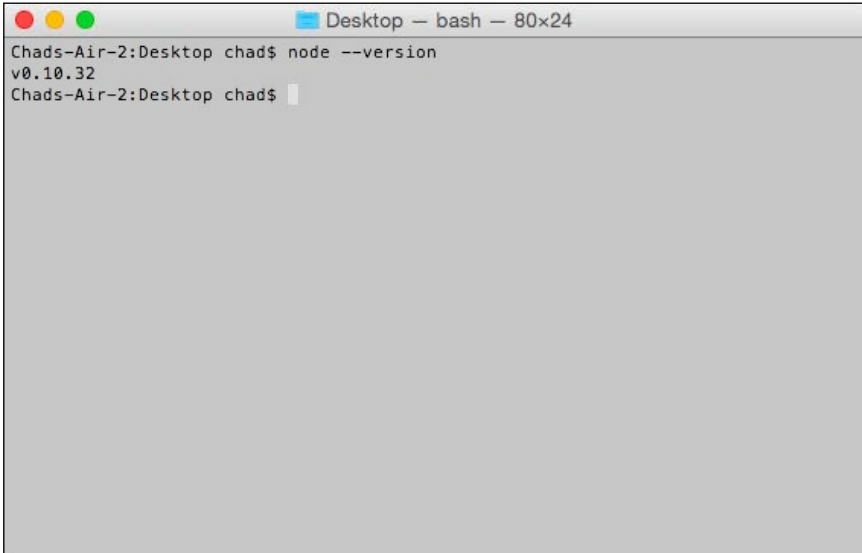
To ensure that Node.js is properly installed, we want to check two things. The first thing is to check whether Node.js works in the Terminal. To validate the installation, we will check the current version of Node.js that's installed.

First, let's open Terminal (or Command Prompt, if using Windows), and insert the `node --version` command as shown in the following screenshot, and press *Enter*:



```
Desktop — bash — 80x24
Chads-Air-2:Desktop chad$ node --version
```

If successful, we should see the version number (in my case, it's `v0.10.32`; your version may be greater than my version number when attempting this) as shown on the next line in the Terminal in the following screenshot:



```
Desktop — bash — 80x24
Chads-Air-2:Desktop chad$ node --version
v0.10.32
Chads-Air-2:Desktop chad$
```

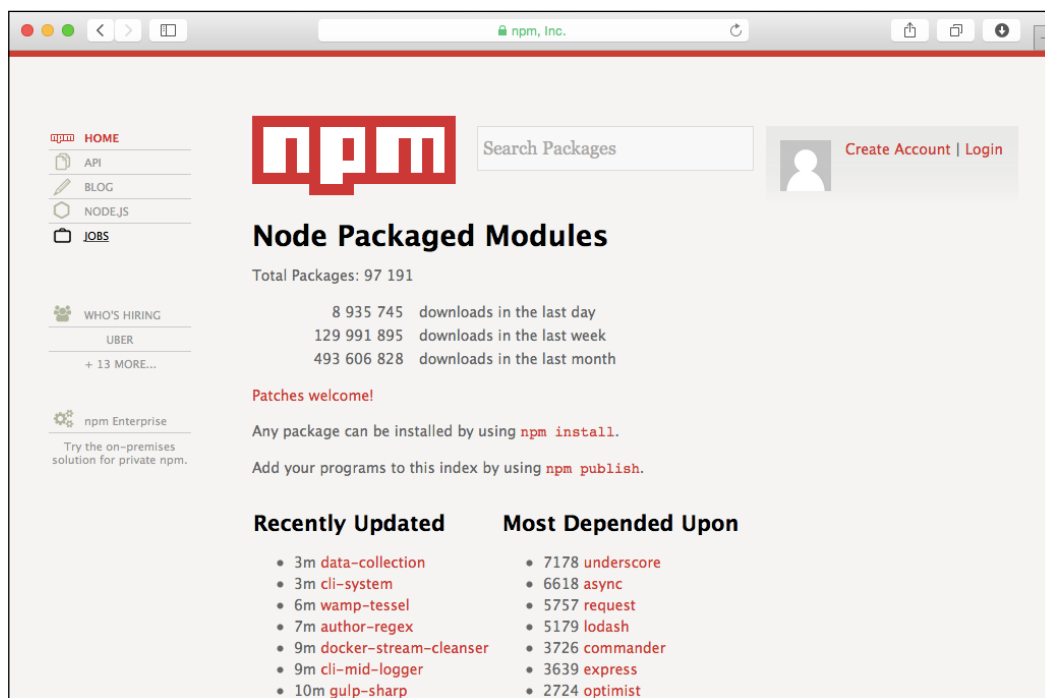
Testing Node Package Manager's installation

Great job! Now, the next thing to be checked for a full installation is whether Node Package Manager is installed as well. Before testing, let me explain what Node Package Manager is, especially to those who might not know what it is and why we need it.

About Node Package Manager

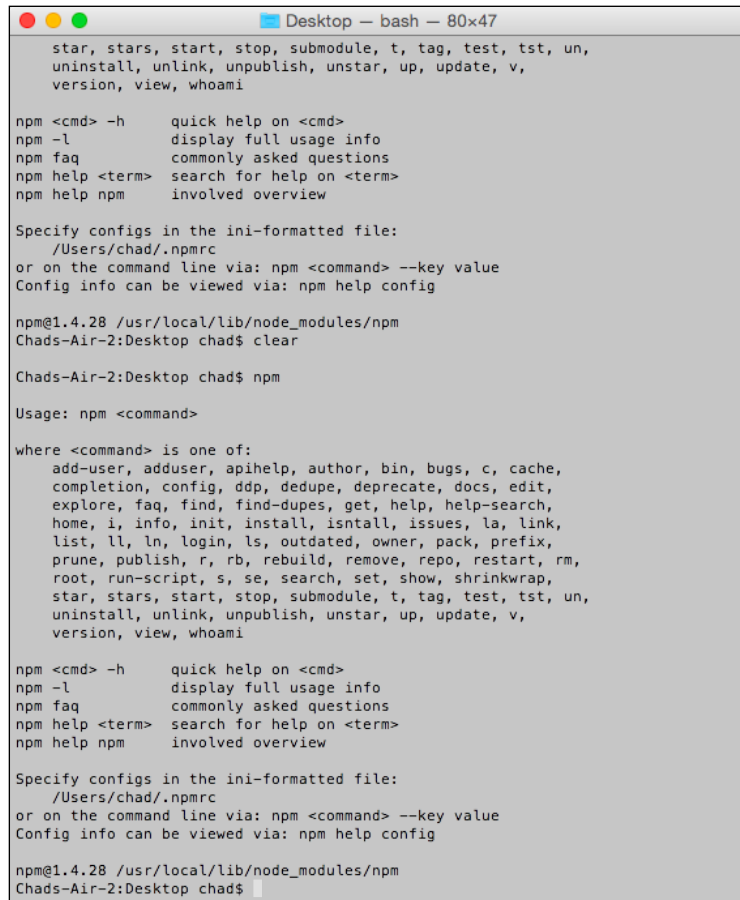
The **Node Package Manager (NPM)** connects to the NPM registry, an online repository of software libraries for Node.js. By using NPM, we can quickly set up a JavaScript build system and install libraries for our HTML-based JavaScript projects automatically, which allows us to ensure that our JavaScript libraries are up-to-date with the latest version of each library.

NPM also has a website you can use to research various JavaScript libraries at <https://www.npmjs.org>. This is also shown in the following screenshot:



Checking NPM installation in the Terminal

Now, to check our installation of NPM, we are going to call NPM directly, which should return a help directory for the NPM module installed. To do this, simply open the Terminal and insert the `npm` command. Now, we should see our Terminal window populating with a bunch of NPM help documentation and sample terminal commands, as shown in the following screenshot:

A screenshot of a terminal window titled "Desktop - bash - 80x47". The terminal displays the output of the `npm` command, which shows a list of available commands and their descriptions. The output is as follows:

```
star, stars, start, stop, submodule, t, tag, test, tst, un,
uninstall, unlink, unpublish, unstar, up, update, v,
version, view, whoami

npm <cmd> -h      quick help on <cmd>
npm -l           display full usage info
npm faq          commonly asked questions
npm help <term>  search for help on <term>
npm help npm     involved overview

Specify configs in the ini-formatted file:
  /Users/chad/.npmrc
or on the command line via: npm <command> --key value
Config info can be viewed via: npm help config

npm@1.4.28 /usr/local/lib/node_modules/npm
Chads-Air-2:Desktop chad$ clear

Chads-Air-2:Desktop chad$ npm

Usage: npm <command>

where <command> is one of:
  add-user, adduser, apihelp, author, bin, bugs, c, cache,
  completion, config, ddp, dedupe, deprecate, docs, edit,
  explore, faq, find, find-dupes, get, help, help-search,
  home, i, info, init, install, isntall, issues, la, link,
  list, ll, ln, login, ls, outdated, owner, pack, prefix,
  prune, publish, r, rb, rebuild, remove, repo, restart, rm,
  root, run-script, s, se, search, set, show, shrinkwrap,
  star, stars, start, stop, submodule, t, tag, test, tst, un,
  uninstall, unlink, unpublish, unstar, up, update, v,
  version, view, whoami

npm <cmd> -h      quick help on <cmd>
npm -l           display full usage info
npm faq          commonly asked questions
npm help <term>  search for help on <term>
npm help npm     involved overview

Specify configs in the ini-formatted file:
  /Users/chad/.npmrc
or on the command line via: npm <command> --key value
Config info can be viewed via: npm help config

npm@1.4.28 /usr/local/lib/node_modules/npm
Chads-Air-2:Desktop chad$
```

The basics of using NPM

Using NPM is a pretty easy process to learn. The first thing that we need to do before setting up NPM for a project is to create a project root directory; I'm going to mark this as `npm_01` for the first project, but you can name your root as whatever you would like. Now, we are going to open Terminal and change our `bash` directory to match the path to the directory I created.

To change your working directory in the Terminal, the command is Change Directory or `cd`. Using `cd` is pretty easy; simply type the following command:

```
cd [~/path/to/project_dir]
```

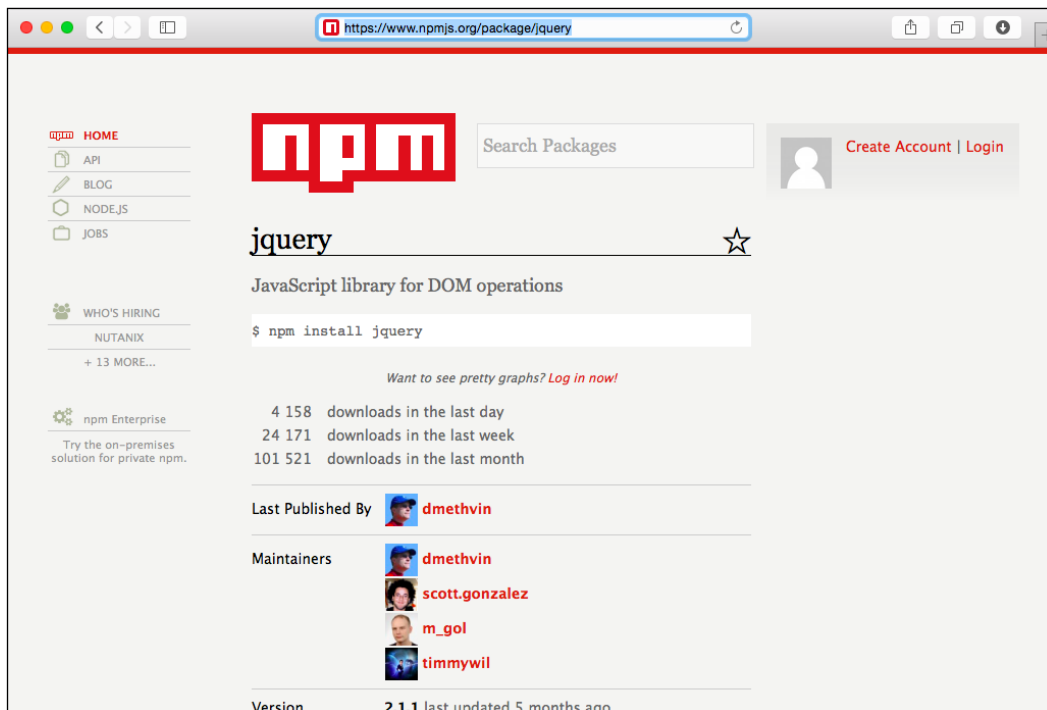
A few things to note here are that the Terminal always points to your user home directory on Mac and Linux, and the tilda key (or `~`) is a shortcut to point to your path. For example, if your folder is in your documents directory under your username, an example path using `cd` would be `cd ~/Documents/[your_project_path]`.




If the Terminal is getting cluttered with information, you can use the "clear" command to clean your terminal's contents, without changing your directory.

Installing jQuery with NPM

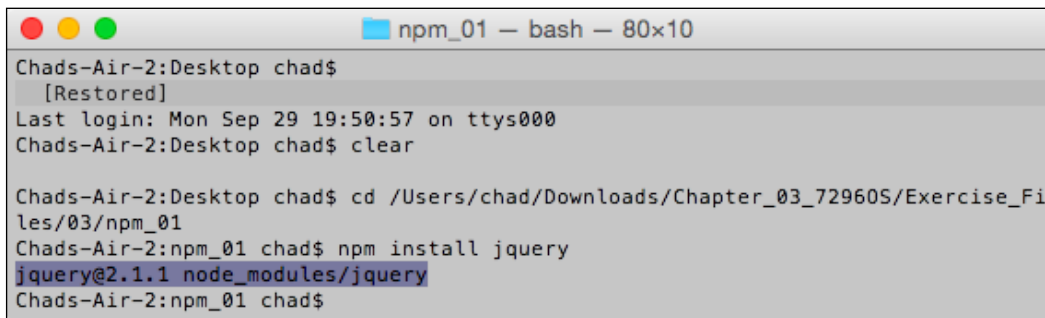
A common JavaScript library is jQuery, a very popular library on NPM. We can even check out its repository information on [npmjs.org](https://www.npmjs.org), found at <https://www.npmjs.org/package/jquery>.



If we look at this page, we can see a command for our terminal, `npm install jQuery`. So, let's type that into our terminal and press *Enter* to see what happens.

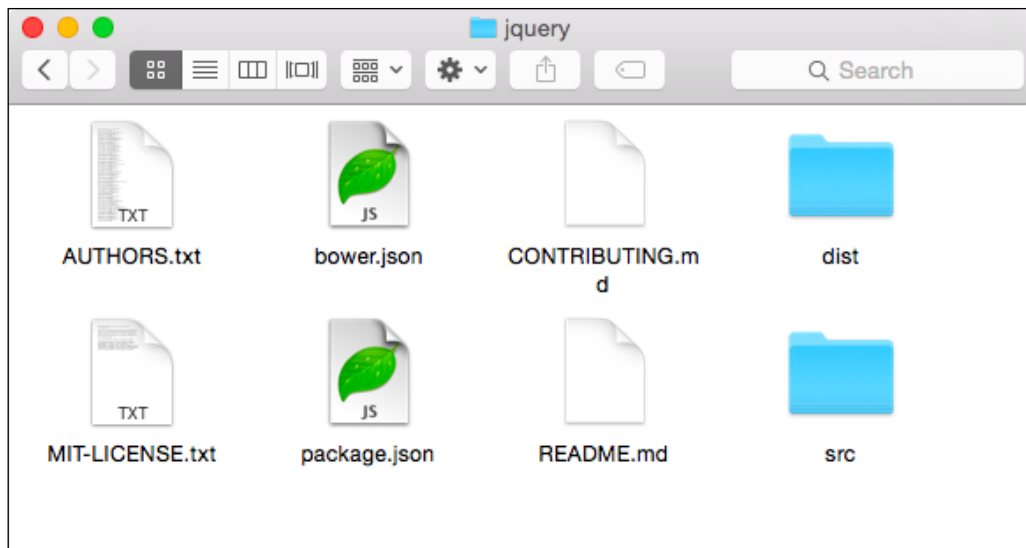
[ If you're a Mac or Linux user, you can drag-and-drop a folder into the Terminal, and it will auto-write the path of your folder for you after you type the `cd` command.]

In the Terminal, it looks like some files were downloaded, as indicated in the following screenshot:



```
Chads-Air-2:Desktop chad$  
[Restored]  
Last login: Mon Sep 29 19:50:57 on ttys000  
Chads-Air-2:Desktop chad$ clear  
  
Chads-Air-2:Desktop chad$ cd /Users/chad/Downloads/Chapter_03_72960S/Exercise_Files/03/npm_01  
Chads-Air-2:npm_01 chad$ npm install jquery  
jquery@2.1.1 node_modules/jquery  
Chads-Air-2:npm_01 chad$
```

Now, if we open our project directory, we can see that a new folder named `node_modules` has been created. Inside this folder, another folder is created named `jquery`. The contents of the `jquery` folder are shown in the following screenshot:



Inside the `jquery` folder, there are some interesting files. We have a `README.md` (`.md` is short for markdown, a type of text format) file explaining jQuery.

The folder has two JSON files, one called `bower.json` and another called `package.json`. The `package.json` file handles the NPM package information while the `bower.json` file handles any dependent packages and notifies NPM to include those as well on installation request.

If you're wondering what the `bower.json` file does, it's essentially another way to update source code from a repository. Like the NPM registry, the `bower.json` file uses its own; the difference is that it can use a JSON file in a project and update based on the setting stored in the JSON file.

Lastly, the most important two folders are the `src` folder (or source folder) and the `dist` folder (or distribution folder). This file structure is a common convention for NPM, where the source of a project with debug information is saved in the `src` folder, while the final tested output lives in the `dist` folder.

Since we aren't debugging the source for jQuery, all we really need to worry about is the `dist` folder, where we can find the `jquery.js` file and the `jquery.min.js` file—the same library files typically used in jQuery projects. It's important to know this for our build systems as we will want to copy those into the distribution folder for our build system.

Setting up our build system

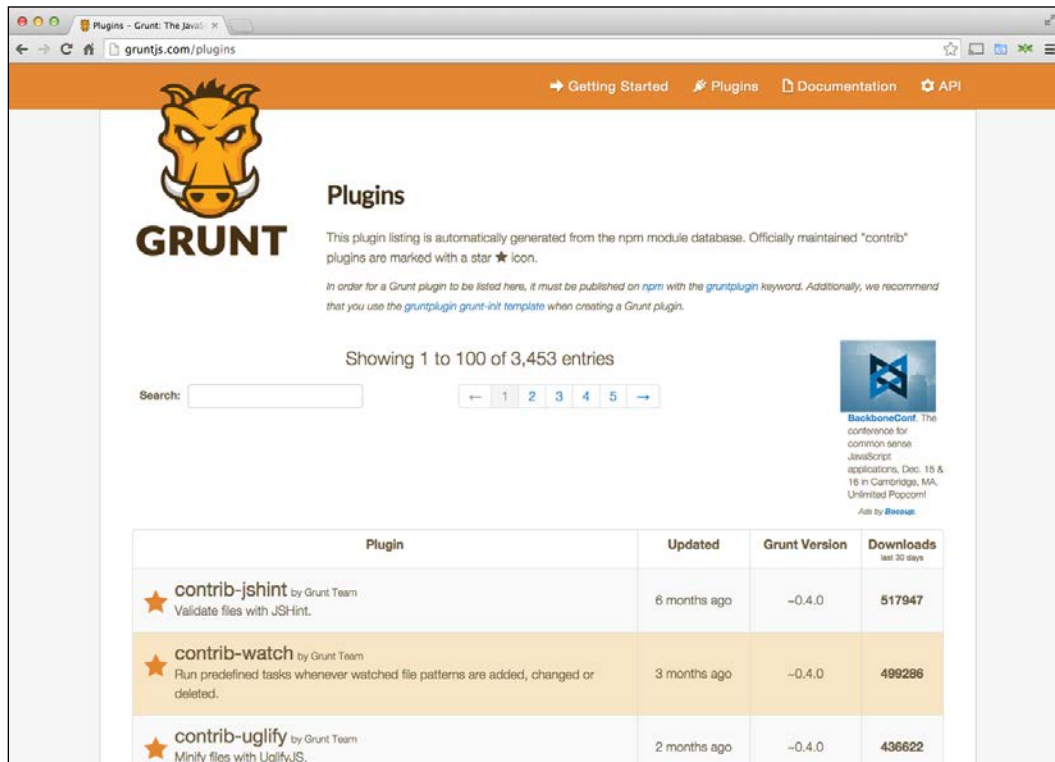
Now that we have learned the basics of Node.js and NPM, let's actually build a build system. We will want to point our Terminal to our project's root directory, and then we will want to install our build system (also called Task Runner).

About Grunt.js and Gulp.js

Node.js build systems fall within two major build system libraries: Grunt and Gulp. Grunt is, in many cases, the default build system for Node.js projects.

Grunt Task Runner

Grunt was designed originally for automating tasks in JavaScript and, web development and due to its availability, many developers have created plugins; you can view them in Grunt's plugin repository shown in the following screenshot:

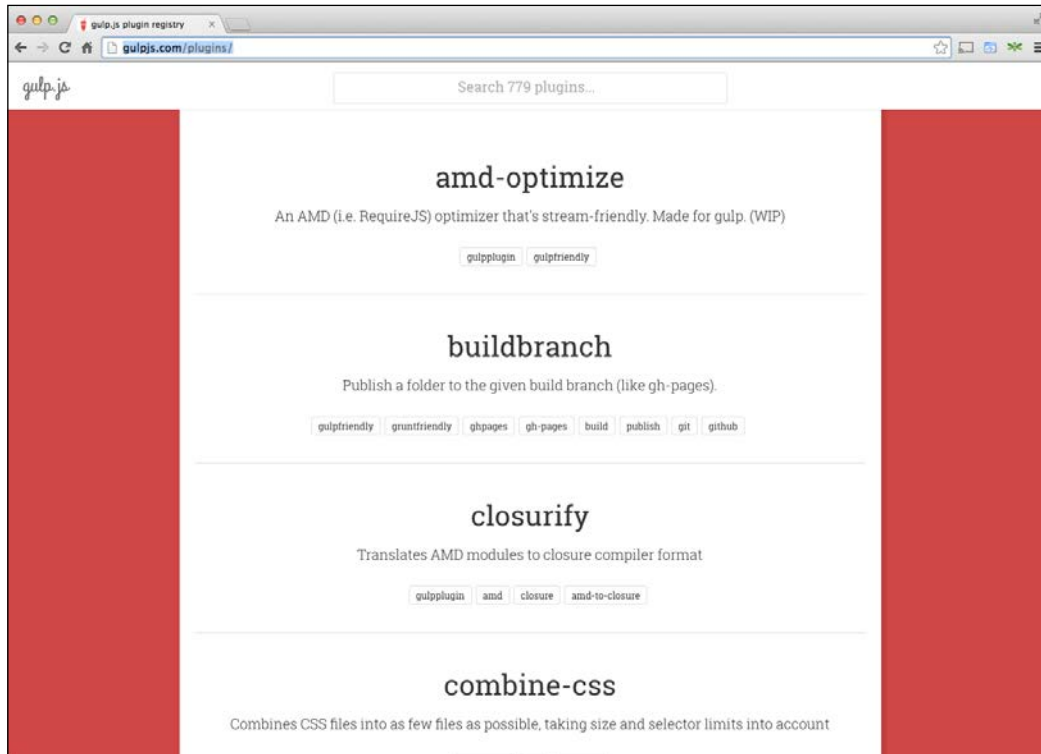
A screenshot of a web browser showing the Grunt.js Plugins page. The page has an orange header with navigation links: Getting Started, Plugins, Documentation, and API. The main content area features the Grunt logo (a stylized orange bull head) and the word "GRUNT" in large, bold, black letters. Below the logo, there is a "Plugins" section with a description: "This plugin listing is automatically generated from the npm module database. Officially maintained 'contrib' plugins are marked with a star ★ icon." It also includes a note about the requirements for a plugin to be listed. A search bar and pagination controls are present. A table lists the top plugins, including contrib-jshint, contrib-watch, and contrib-uglify. A sidebar on the right promotes BackboneConf.

Plugin	Updated	Grunt Version	Downloads last 30 days
★ contrib-jshint by Grunt Team Validate files with JSHint.	6 months ago	~0.4.0	517947
★ contrib-watch by Grunt Team Run predefined tasks whenever watched file patterns are added, changed or deleted.	3 months ago	~0.4.0	499286
★ contrib-uglify by Grunt Team Minify files with UglifyJS.	2 months ago	~0.4.0	436622

About Gulp

Gulp is another build system for Node.js; the advantage of using Gulp is that it's asynchronous, and typically runs automated tasks much faster than Grunt. Since this book is all about performance, we will use Gulp for our build system as an example. That doesn't mean that Grunt is bad; it can create build systems as well as Gulp.js, but it may not be as fast as Gulp.

Like Grunt, Gulp also has a plugin reference page found at <http://gulpjs.com/plugins/>, and shown in the following screenshot:

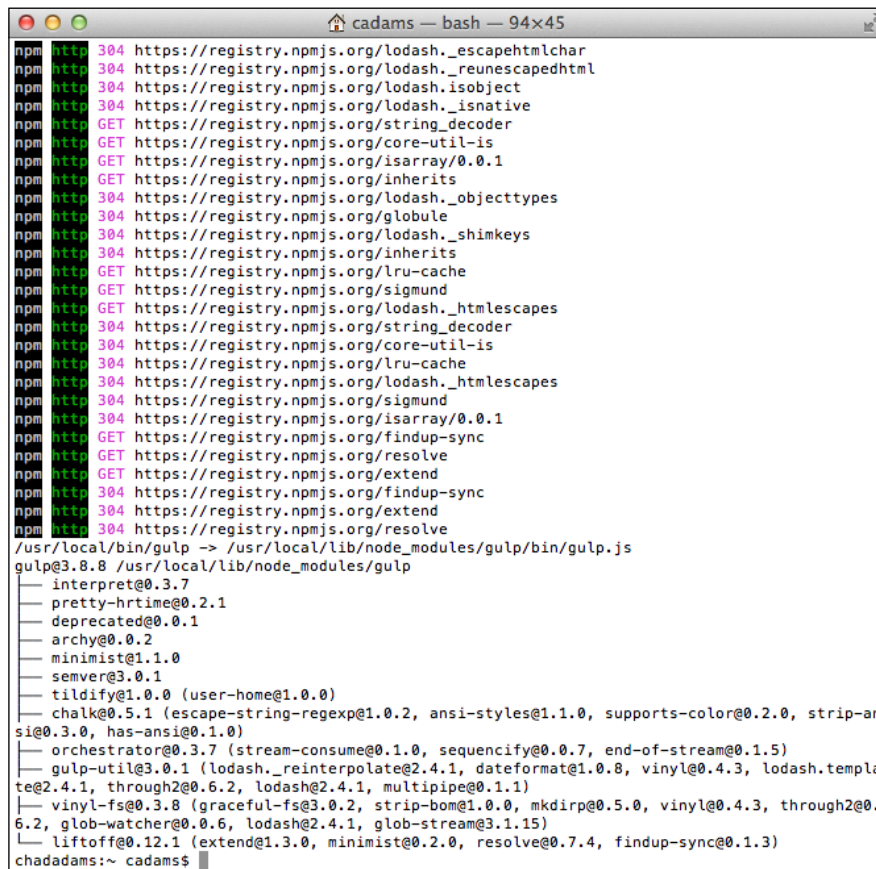


Installing Gulp

To install Gulp, we will open our terminal and type the following into our prompt:

```
sudo npm install --global gulp
```

This will install Gulp globally to our Node.js and NPM resource path. If we are running a Windows system, `sudo` isn't in Window's Shell, so we will need to run the Command Prompt as Administrator. Now, if everything is successful, we should see a bunch of URL requests for files, and our Terminal should return to the prompt shown in the following screenshot:



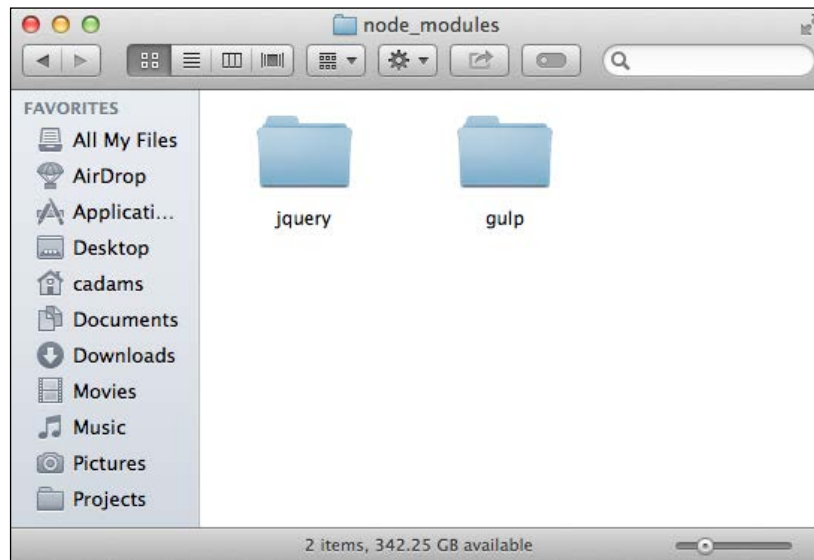
```
cadams — bash — 94x45
npm http 304 https://registry.npmjs.org/lodash._escapehtmlchar
npm http 304 https://registry.npmjs.org/lodash._reunescapehtml
npm http 304 https://registry.npmjs.org/lodash.isobject
npm http 304 https://registry.npmjs.org/lodash.isnative
npm http GET https://registry.npmjs.org/string_decoder
npm http GET https://registry.npmjs.org/core-util-is
npm http GET https://registry.npmjs.org/isarray/0.0.1
npm http GET https://registry.npmjs.org/inherits
npm http 304 https://registry.npmjs.org/lodash._objecttypes
npm http 304 https://registry.npmjs.org/globule
npm http 304 https://registry.npmjs.org/lodash._shimkeys
npm http 304 https://registry.npmjs.org/inherits
npm http GET https://registry.npmjs.org/lru-cache
npm http GET https://registry.npmjs.org/sigmund
npm http GET https://registry.npmjs.org/lodash._htmlescapes
npm http 304 https://registry.npmjs.org/string_decoder
npm http 304 https://registry.npmjs.org/core-util-is
npm http 304 https://registry.npmjs.org/lru-cache
npm http 304 https://registry.npmjs.org/lodash._htmlescapes
npm http 304 https://registry.npmjs.org/sigmund
npm http 304 https://registry.npmjs.org/isarray/0.0.1
npm http GET https://registry.npmjs.org/findup-sync
npm http GET https://registry.npmjs.org/resolve
npm http GET https://registry.npmjs.org/extend
npm http 304 https://registry.npmjs.org/findup-sync
npm http 304 https://registry.npmjs.org/extend
npm http 304 https://registry.npmjs.org/resolve
/usr/local/bin/gulp -> /usr/local/lib/node_modules/gulp/bin/gulp.js
gulp@3.8.8 /usr/local/lib/node_modules/gulp
├── interpret@0.3.7
├── pretty-hrtime@0.2.1
├── deprecated@0.0.1
├── archy@0.0.2
├── minimist@1.1.0
├── semver@3.0.1
├── tildify@1.0.0 (user-home@1.0.0)
├── chalk@0.5.1 (escape-string-regexp@1.0.2, ansi-styles@1.1.0, supports-color@0.2.0, strip-ansi@0.3.0, has-ansi@0.1.0)
├── orchestrator@0.3.7 (stream-consume@0.1.0, sequencify@0.0.7, end-of-stream@0.1.5)
├── gulp-util@3.0.1 (lodash._reinterpolate@2.4.1, dateFormat@1.0.8, vinyl@0.4.3, lodash.templates@2.4.1, through2@0.6.2, lodash@2.4.1, multipipe@0.1.1)
├── vinyl-fs@0.3.8 (graceful-fs@3.0.2, strip-bom@1.0.0, mkdirp@0.5.0, vinyl@0.4.3, through2@0.6.2, glob-watcher@0.0.6, lodash@2.4.1, glob-stream@3.1.15)
├── liftoff@0.12.1 (extend@1.3.0, minimist@0.2.0, resolve@0.7.4, findup-sync@0.1.3)
chadadams:~ cadams$
```

With our global ("global" meaning installed for all folders in our system) dependencies for Gulp installed, we can install our developer dependencies, which allow our build system to be more portable when uploading to a source control. Essentially, these dependencies need to be in our root project file to enable our build system to run in our project directory.

We can do it by typing the following code into our Terminal (again, `sudo` for Mac/Linux users and **Run as Administrator** for Windows users):

```
sudo npm install --save-dev gulp
```

If successful, your `bash` prompt should show again, pulling many URL sources and installing them to your project's `node_modules` directory under `gulp` as shown in the following screenshot:



Creating a gulpfile

A `gulpfile` is a file that Gulp checks for to run a list of tasks at the root of our project directory. To create one, we will create a simple JavaScript file called `gulpfile.js` (note the case of the filename). Inside the file we are going to reference Gulp as a variable and create a default task, called `Default`.

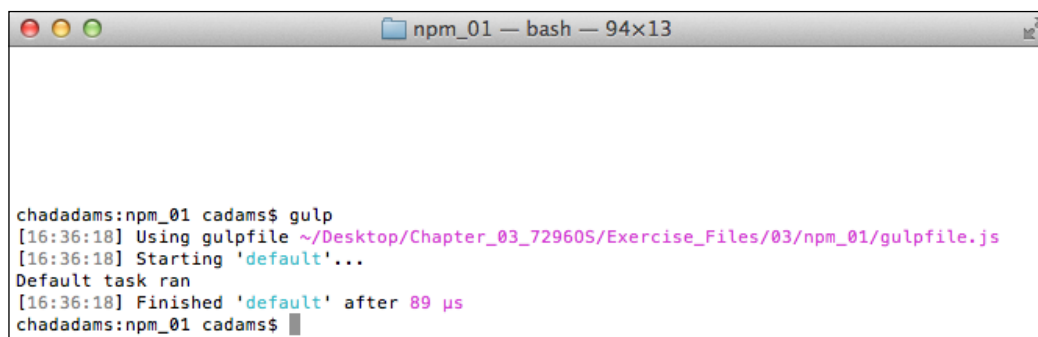
This is the main task we need to run for every `gulpfile.js`; inside it, we can include other tasks or output log message, just like in a web browser. As a simple code example for a Gulp task is shown in the following screenshot:



```
1 var gulp = require('gulp');
2
3 gulp.task('default', function() {
4   /* Required 'default' task. */
5   console.log('Default task ran');
6 }
7 });
```

Running a Gulp project

Running a Gulp project is easy. In your project's root directory, type `gulp` in the Terminal and press *Enter*. You should see the output in your terminal as shown in the following screenshot:



```
chadadams:npm_01 cadams$ gulp
[16:36:18] Using gulpfile ~/Desktop/Chapter_03_729605/Exercise_Files/03/npm_01/gulpfile.js
[16:36:18] Starting 'default'...
Default task ran
[16:36:18] Finished 'default' after 89 μs
chadadams:npm_01 cadams$
```

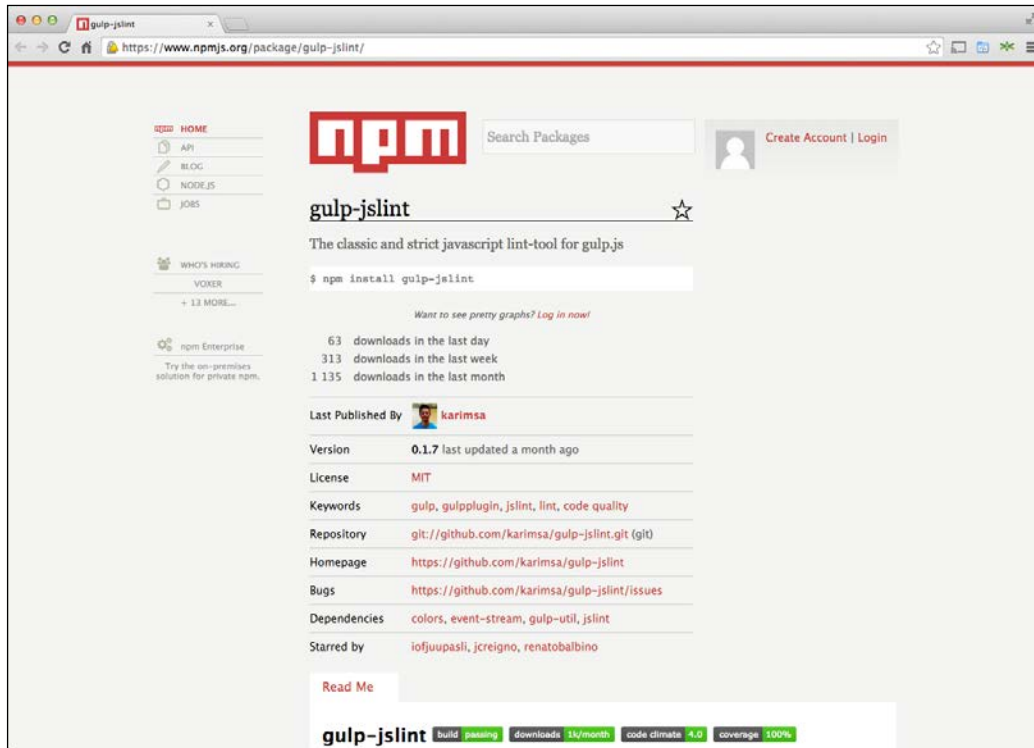
That's nice. If we look at the fourth line of our Terminal's output, we should see our output message as **Default task ran**. Good job! This is the same `console.log` message we created for our `Default` task in our `gulpfile.js`.

So you may ask, How did all this help optimize the JavaScript code? Well, if you remember *Chapter 2, Increasing Code Performance with JSLint*, we used JSLint to review JavaScript code, make improvements, and optimize files. What if we could run that test tool, while copying files and minifying (or even testing minified code) through JSLint? Well, we can, and that's the point of using build systems.

With a build system, we are improving it and optimizing our code before modifying it, even before it is deployed out as a web application.

Integrating JSLint into Gulp

Earlier, we talked about Gulp's plugin page; well one of those plugins is a JSLint plugin, and the installation process is pretty easy. First, check out the JSLint plugin page found at <https://www.npmjs.org/package/gulp-jshint/>, and as shown in the following figure:



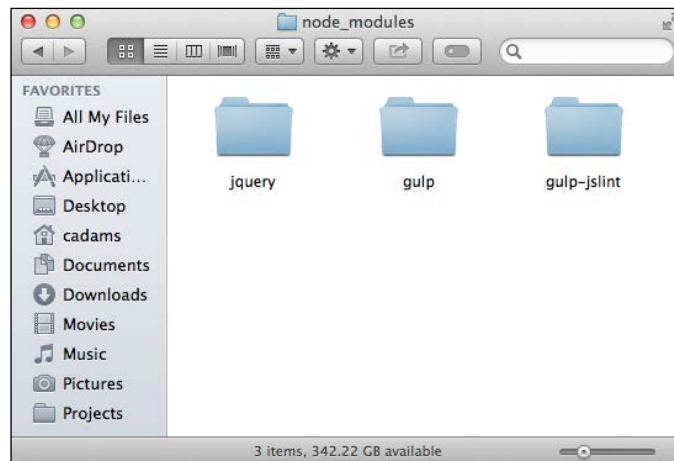
So, in the same way we installed Gulp, we will run the `npm` command shown on the page, but will include `sudo` for administrator permissions and the `-g` command. This is a global flag to install JSLint to the full system, as follows:

```
sudo npm install -g gulp-jshint
```

Next, we will install the developer dependencies for our project, so again we will point to our root project directory in the Terminal and then type our `npm` command but with the `-save-dev` flag this time, as follows:

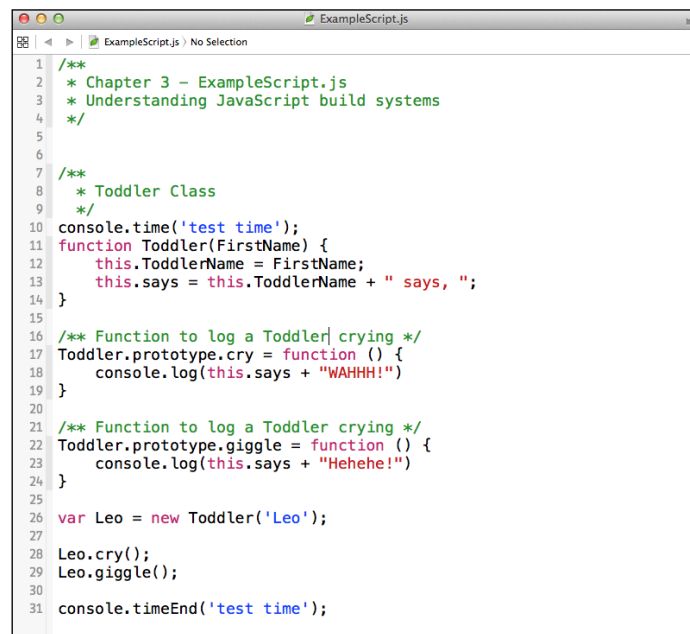
```
sudo npm install -save-dev gulp-jshint
```


To verify the installation, we can check the `node_modules` folder in our project directory, and see the `gulp-jshint` folder, as shown in the following screenshot:

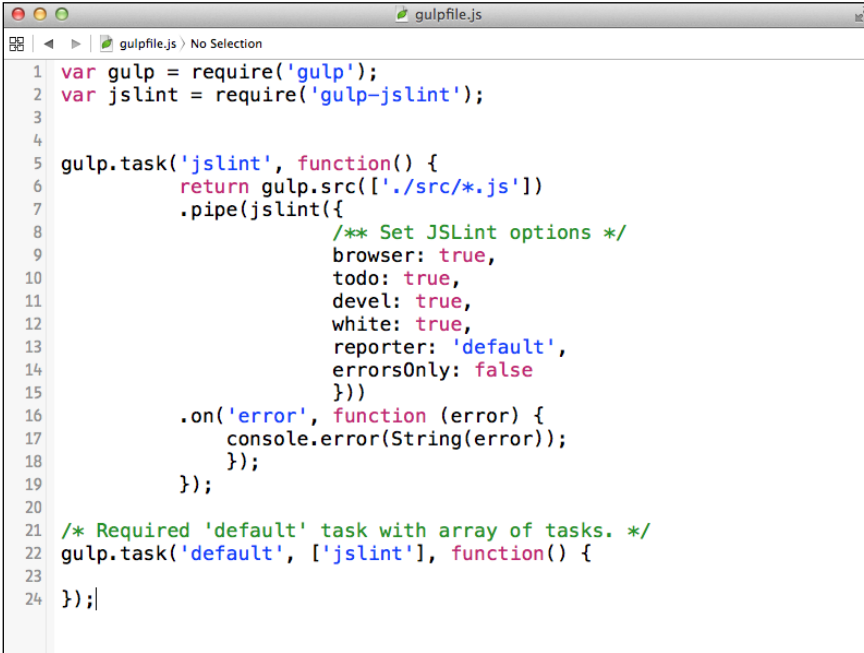


Testing our example file

Now, our build system needs a source file, and I've written an example while adding it to a new `src` project directory created in Finder. I haven't tested this yet, and it's shown in the following screenshot:



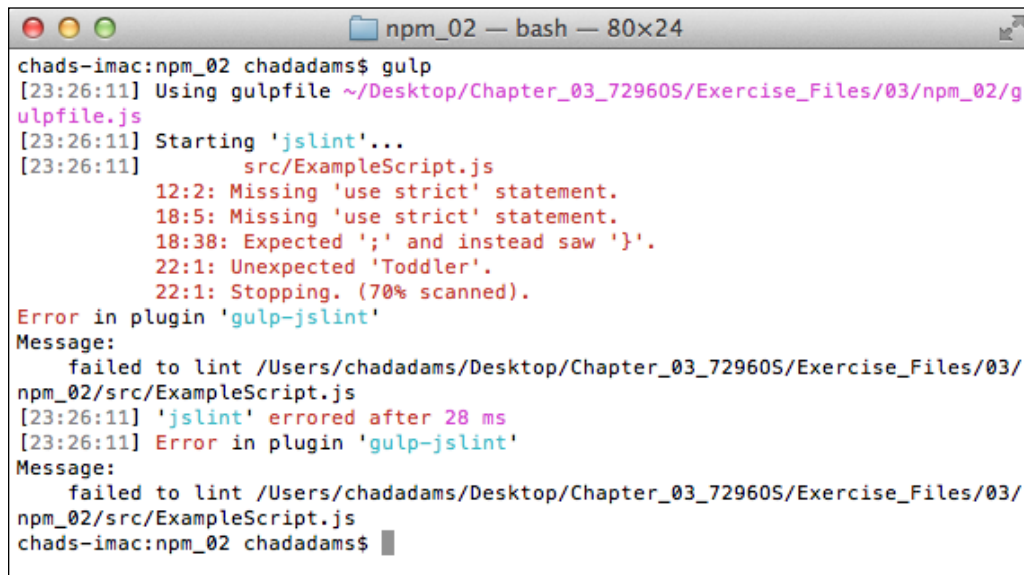
So we have a simple Toddler JavaScript class, and it displays messages based on the prototype functions called; it's pretty basic and it does have some intended errors, so let's find these. Let's go back to `gulpfile.js`; I've updated it with some JSLint examples using the same common options enabled by us and mentioned in *Chapter 2, Increasing Code Performance with JSLint*. Take a look at the updated `gulpfile.js` file, shown in the next screenshot:



```
1 var gulp = require('gulp');
2 var jshint = require('gulp-jshint');
3
4
5 gulp.task('jslint', function() {
6     return gulp.src(['./src/*.js'])
7         .pipe(jshint({
8             /* Set JSLint options */
9             browser: true,
10            todo: true,
11            devel: true,
12            white: true,
13            reporter: 'default',
14            errorsOnly: false
15        })))
16     .on('error', function (error) {
17         console.error(String(error));
18     });
19 });
20
21 /* Required 'default' task with array of tasks. */
22 gulp.task('default', ['jslint'], function() {
23
24 });
```

On lines 6 and 7, we can see conventions such as, `gulp.src()` and `pipe()`. The `src` function is a Gulp-specific function that sets the source file or files using a JavaScript array; the `pipe` function, which is also Gulp-related, allows us to create a list of tasks that will take the source files from `gulp.src()` and *pipe* them through our build system. Lines 5 to 19 here show a new `gulp.task` called JSLint. If we look at lines 9 to 12, we can see the same options used from `JSLint.com`; the option names can be found under JSLint Directives at the bottom of the page when we select different options on the page.

On line 22, we added an array after our *default* task, adding our *JSLint* task name into an array. We can add multiple tasks here, but for now we just need the lint task. Now let's run the script and check our terminal.

A terminal window titled 'npm_02 — bash — 80x24' showing the execution of the 'gulp' command. The output shows the gulpfile being used, the start of the 'jslint' task, and a list of linting errors for 'src/ExampleScript.js'. The errors include missing 'use strict' statements, an unexpected semicolon, and an unexpected word 'Toddler'. The terminal also shows error messages from the 'gulp-jshint' plugin.

```
chads-imac:npm_02 chadadams$ gulp
[23:26:11] Using gulpfile ~/Desktop/Chapter_03_72960S/Exercise_Files/03/npm_02/gulpfile.js
[23:26:11] Starting 'jshint'...
[23:26:11]      src/ExampleScript.js
      12:2: Missing 'use strict' statement.
      18:5: Missing 'use strict' statement.
      18:38: Expected ';' and instead saw '}'.
      22:1: Unexpected 'Toddler'.
      22:1: Stopping. (70% scanned).
Error in plugin 'gulp-jshint'
Message:
    failed to lint /Users/chadadams/Desktop/Chapter_03_72960S/Exercise_Files/03/npm_02/src/ExampleScript.js
[23:26:11] 'jshint' errored after 28 ms
[23:26:11] Error in plugin 'gulp-jshint'
Message:
    failed to lint /Users/chadadams/Desktop/Chapter_03_72960S/Exercise_Files/03/npm_02/src/ExampleScript.js
chads-imac:npm_02 chadadams$
```

Excellent! The red lines shown in the terminal report errors with the script that gives us our lint feedback in the terminal and, as we can see, we forgot some common things such as using *use strict*, missing semicolons, and so on. So we can see how we can automate the process of testing our code during a build using Node.js and Gulp.

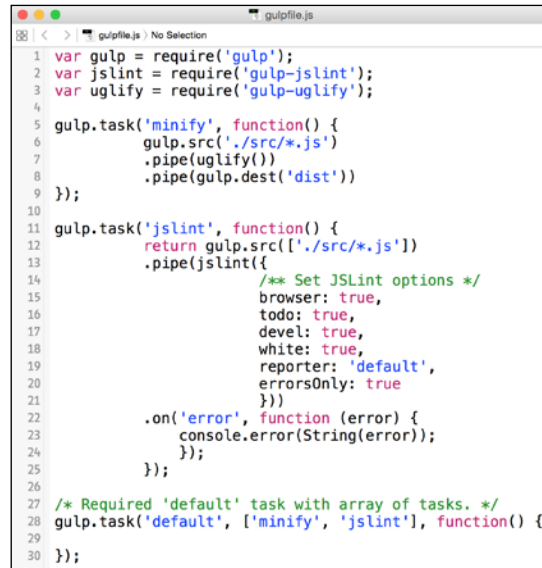
Creating a distribution

Saving the best part for last, let's have Gulp handle minifying the JavaScript source code, copying the output to the *dist* folder, and then linting the output for testing. I've modified the *ExampleScript.js* file to fix most of the issues found earlier.

Now we need to download a minification tool for Gulp called **Uglify**, available at <https://www.npmjs.org/package/gulp-uglify>. It's a common minifier for JavaScript for Gulp-based projects; its installation is easy and follows the same procedure used for installing Gulp itself and JSLint for Gulp. The following command is used for installing the tool:

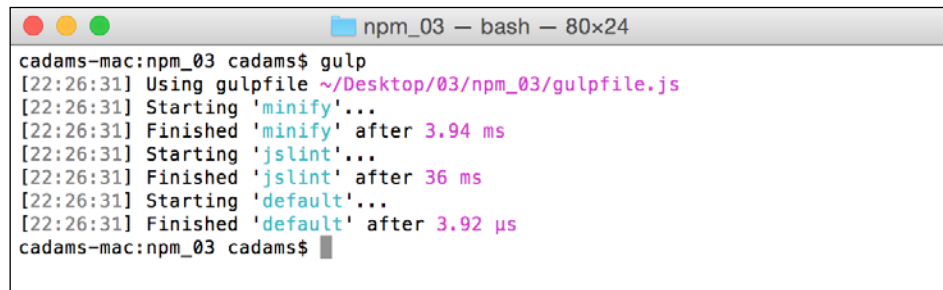
```
sudo npm install --save-dev gulp-uglify
```

Now I've updated our `gulpfile.js` with a new minify task and added it to the array as shown in the following screenshot:



```
1 var gulp = require('gulp');
2 var jslint = require('gulp-jshint');
3 var uglify = require('gulp-uglify');
4
5 gulp.task('minify', function() {
6   gulp.src('./src/*.js')
7     .pipe(uglify())
8     .pipe(gulp.dest('dist'))
9 });
10
11 gulp.task('jslint', function() {
12   return gulp.src(['./src/*.js'])
13     .pipe(jslint({
14       /* Set JSLint options */
15       browser: true,
16       todo: true,
17       devel: true,
18       white: true,
19       reporter: 'default',
20       errorsOnly: true
21     }));
22   .on('error', function (error) {
23     console.error(String(error));
24   });
25 });
26
27 /* Required 'default' task with array of tasks. */
28 gulp.task('default', ['minify', 'jslint'], function() {
29
30 });
```

Now, run Gulp in the Terminal window and notice the output (shown in the next screenshot); in the finder folder, you'll see a brand-new minified file in the `dist` directory of your root project folder, while retaining your developer source and getting performance linting at the same time!



```
cadams-mac:npm_03 cadams$ gulp
[22:26:31] Using gulpfile ~/Desktop/03/npm_03/gulpfile.js
[22:26:31] Starting 'minify'...
[22:26:31] Finished 'minify' after 3.94 ms
[22:26:31] Starting 'jslint'...
[22:26:31] Finished 'jslint' after 36 ms
[22:26:31] Starting 'default'...
[22:26:31] Finished 'default' after 3.92 μs
cadams-mac:npm_03 cadams$
```

Summary

In this chapter, we learned how to create a simple JavaScript build system using Node.js with Gulp. We also explored other plugins and checked out Grunt Task Runner, which works similar to Gulp but contains many more plugins for your work.

Build systems help your performance greatly without much effort; keep in mind that gulp files can be reused for other projects, and so experiment and find out what works best for your projects.

In the next chapter, we will learn tips and tricks on how to use Chrome's **Developer tools** option to better optimize our web application code.

4

Detecting Performance

In this chapter, we'll cover our work environment and the tools needed; we will also cover the features and JavaScript optimization tools found in the Google Chrome Web Inspector, and create some test samples that show us how to use and test JavaScript and HTML page code.

We will cover the following topics in the chapter:

- Web Inspectors in general
- The Elements panel
- The Network panel
- The Timeline panel
- The Profile panel
- The Resources panel
- The Audits panel
- The Console panel

Web Inspectors in general

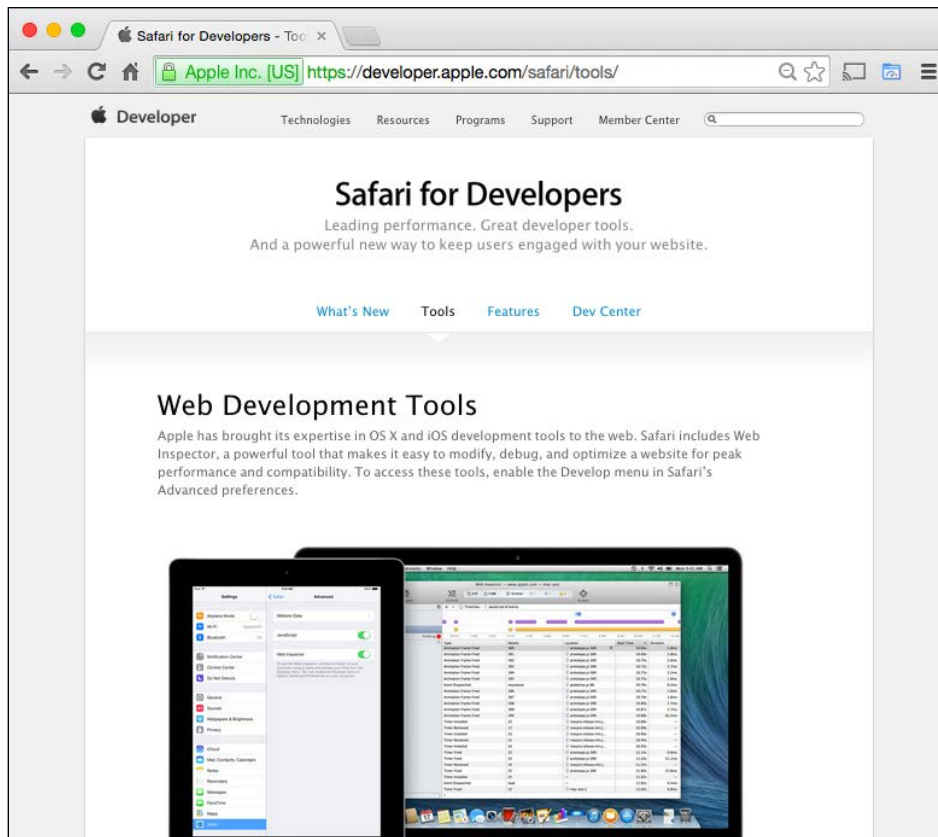
Before diving in to an in-depth exploration of Chrome's Web Inspector, it's important to note that there are many different Web Inspectors for different web browsers, typically developed by the browser's vendor for debugging a web page's application content and performance.

It's important to understand that, for developers to properly debug a web application, they should use the inspector designed for the browser with a detected issue.

The Safari Web Inspector

Apple's Web Inspector is a WebKit-based inspector built for Safari. The Web Inspector is built pretty similar to Chrome's Web Inspector. We will cover more on the Safari **Web Inspector** later in *Chapter 9, Optimizing JavaScript for iOS Hybrid Apps*, mainly because Safari's **Web Inspector** can debug web content in iOS development.

Apple has pretty comprehensive documentation on its tools at <https://developer.apple.com/safari/tools/> and shown in the next screenshot:

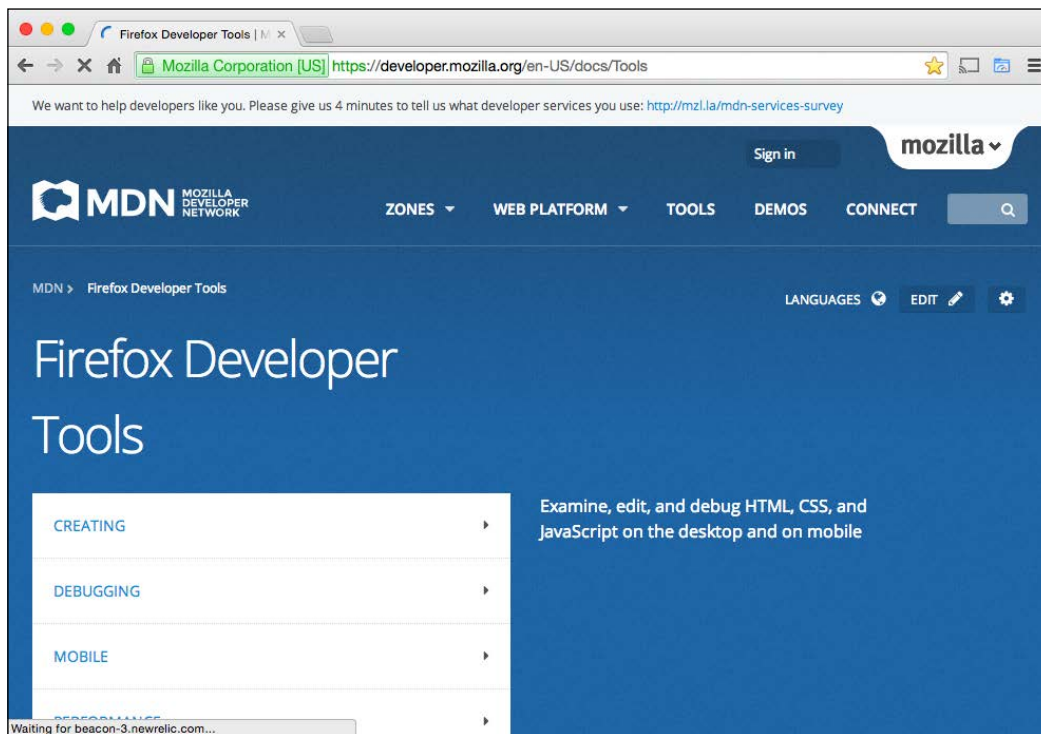


Firefox Developer tools

Mozilla's Firefox web browser also has its own inspector. Originally Firefox was the only browser with an inspector; it was called Firebug, was developed as a plugin, and was not included with the main browser.

Since the advent of Firefox 3, Mozilla developed its own browser inspector not just for their own browser but also as a debug tool for Firefox OS, Mozilla's mobile OS that uses HTML5 for application development. Firefox **Developer** tools also allow debugging for fairly new and even experimental forms of HTML5 and JavaScript development.

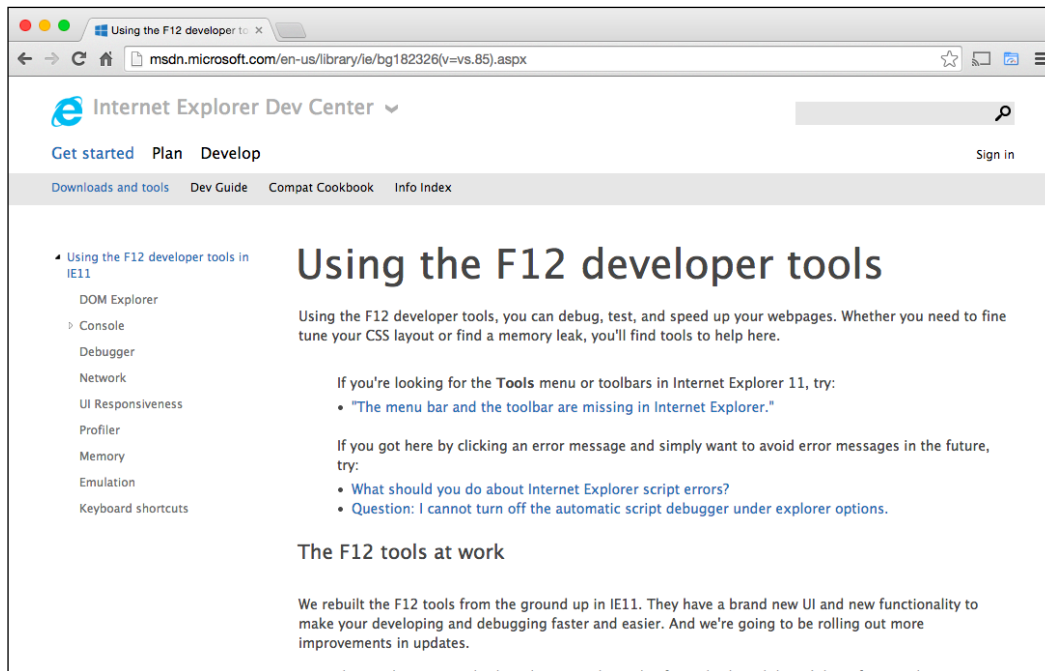
We can check out more information on the types of developers that the Firefox **Developer** tools allow for at Mozilla's Developer Network at <https://developer.mozilla.org/en-US/docs/Tools>, as shown in the next screenshot:



Internet Explorer developer tools

In the past, Internet Explorer was considered the black sheep in the web developer's toolbox. Before the advent of Internet Explorer 11, Microsoft offered a simple DOM inspector plugin for Internet Explorer version 6 and above; though it was very helpful for Internet Explorer's browser issue, its feature set lacked behind other vendors' inspector tools.

Since the release of Internet Explorer 11, Microsoft is positioning itself to support HTML development more than it did in the past, and its new **F12 developer tools** do just that. Most of the features found in the **F12 developer tools** are found just as good as Chrome's **Developer tools** and Safari's **Web Inspector**, with more releases anticipated in future. We can read more on how to use those tools at [http://msdn.microsoft.com/en-us/library/ie/bg182326\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/ie/bg182326(v=vs.85).aspx), as shown in the next screenshot:



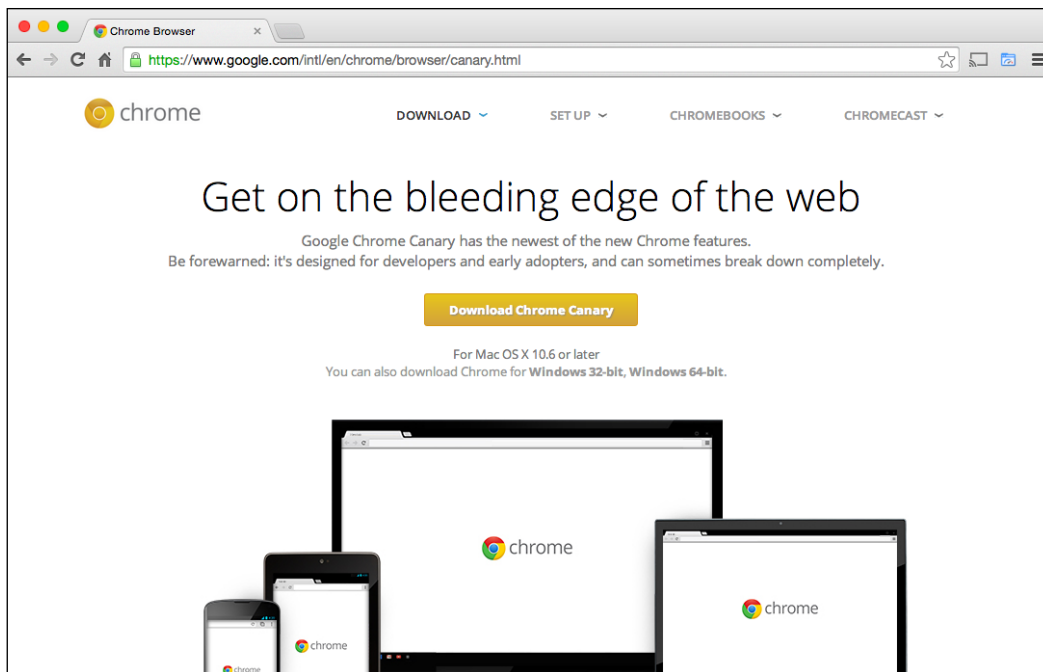
Chrome's Developer tools

Chrome's Inspector was originally developed using the open-source WebKit browser's Web Inspector, which was also used at one point in Apple's Safari. Later, when Chrome decided to fork WebKit into their own browser runtime called Blink, Google rebuilt the Inspector for Blink from the ground up, optimizing the user interface and adding features not found in the open source Webkit Inspector.

Another reason for rebuilding the inspector was the introduction of Chrome for Android and Chrome OS applications. This allows developers to access JavaScript-based console objects specific to development on those platforms. It also features tools to optimize responsive content, and debug mobile content without being on a device.

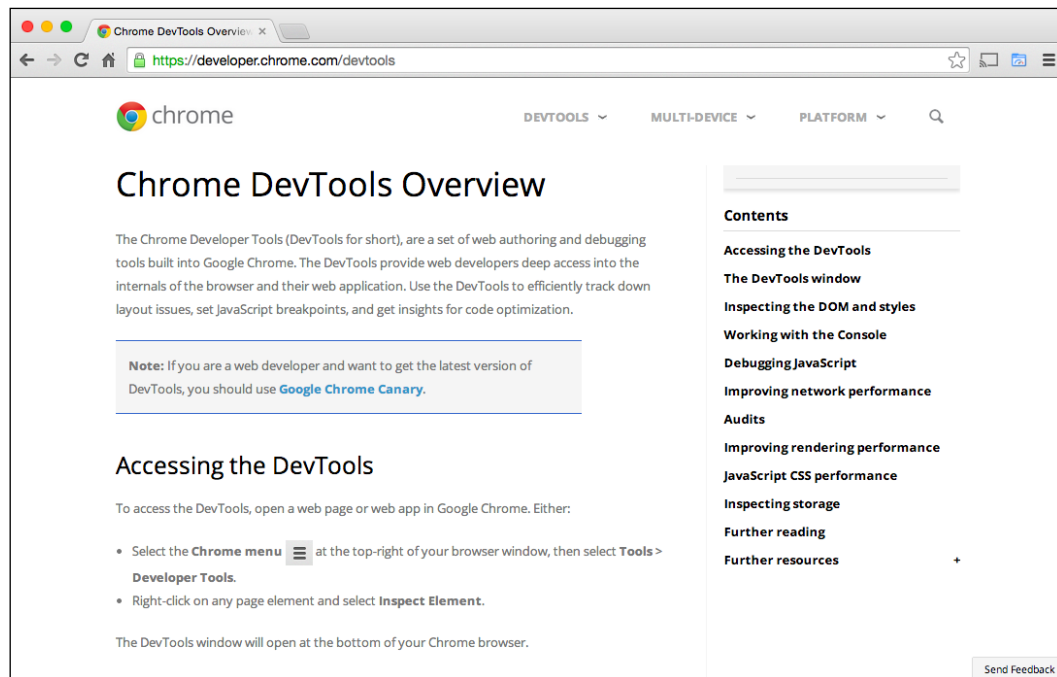
Because of the rich features mentioned here, we will cover how to use the Web Inspector for Chrome. If you're concerned about knowing a feature on another inspector, refer to the previously mentioned links and research a topic listed in this chapter.

Lastly, Chrome's update cycle for new features is pretty frequent and even more so for its beta version of Chrome called Chrome Canary, which is essentially Chrome with experimental features enabled including any early speed improvements for Blink. You can download Canary at <https://www.google.com/intl/en/chrome/browser/canary.html>, as shown in the next screenshot:



Chromium's **Developer tools** include many more advanced features typically found in Firefox **Developer tools**. For this chapter, I'll be using the default Chrome **Developer tools**, but please check out Chromium's **Developer tools** as well to stay informed on what's available in the future.

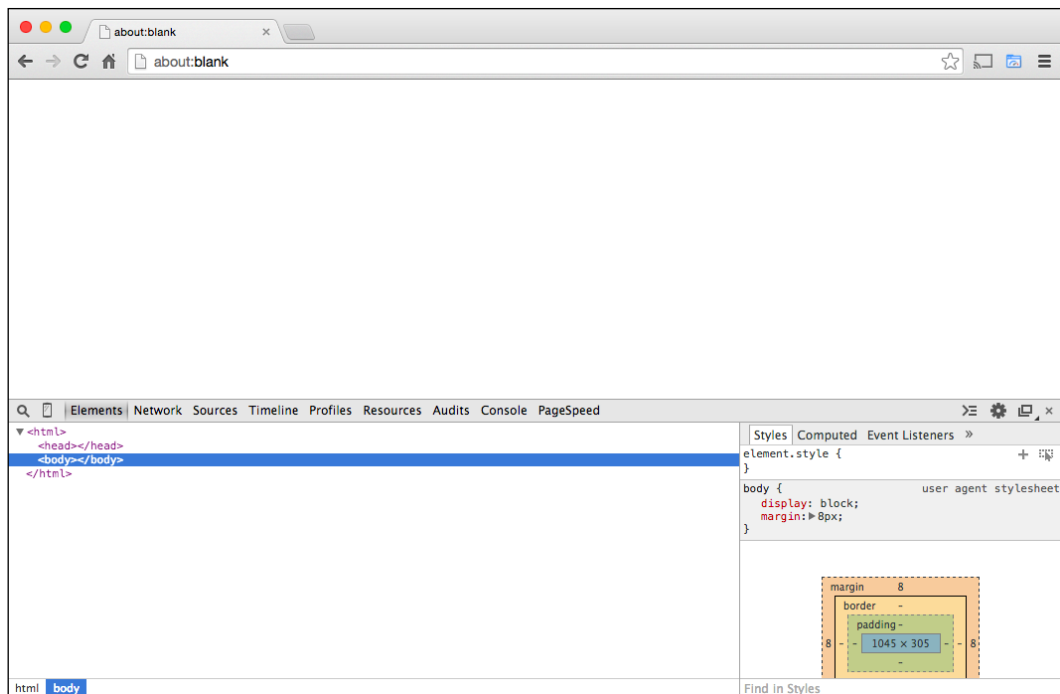
Check out <https://developer.chrome.com/devtools> for **Chrome DevTools Overview**, as shown in the next screenshot:



Getting familiar with Chrome's Developer tools

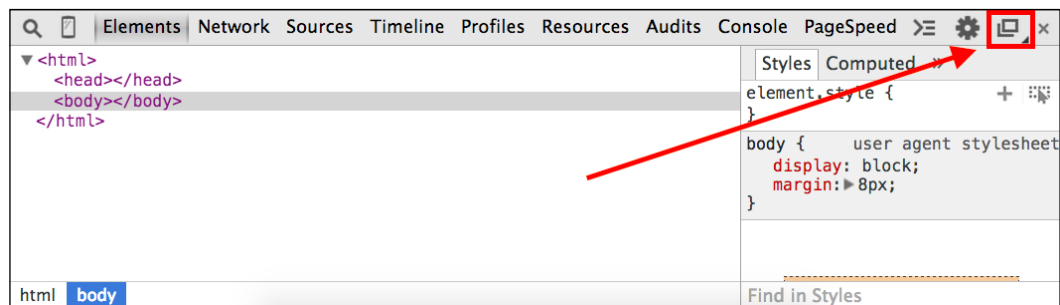
To install Chrome's **Developer tools**, download chrome from <http://www.chrome.com/>, and that's it! Chrome's **Developer tools** are included with Chrome with no extra installation needed.

First, open a new window in Chrome and type `about:blank` in the omnibox (or the address bar). Next, let's open up the **Developer tools** by using the keystrokes *Ctrl + Shift + I* (or *Command + Option + I* on the Mac). We should see a blank screen with **Developer tools** showing up, as shown in the next screenshot:



By default, Chrome's **Developer tools** will either be displayed in dock mode, as shown before, or in its own window; if you want to undock or redock the **Developer tools**, select the dock button.

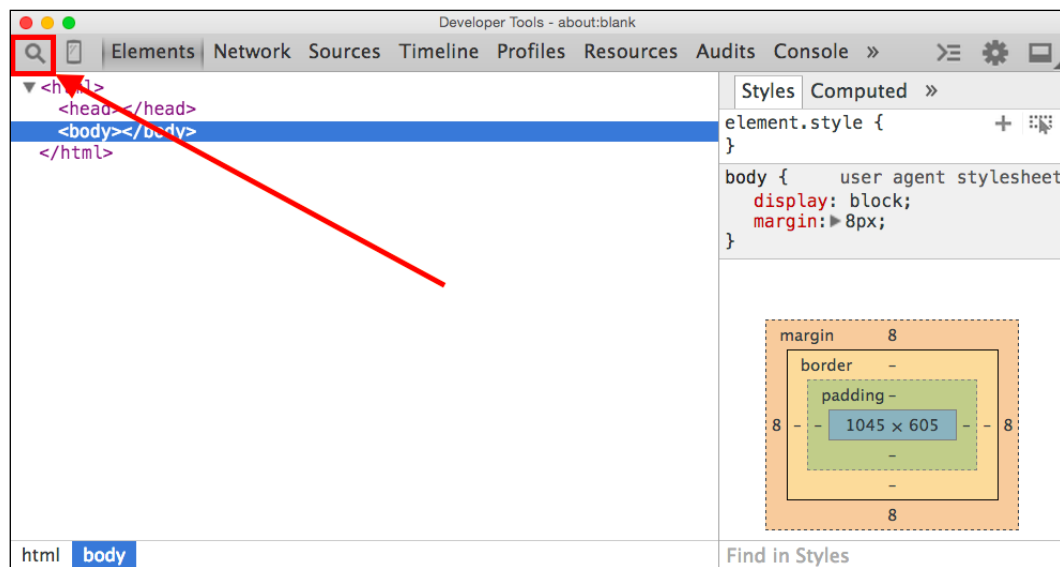
Holding down the dock button allows us to dock the **Developer tools** to the side of the browser window. You can find the dock button flagged in the following screenshot:



The **Developer tools** are broken up into different panels, that are shown at the top of the window, each panel containing different features and debugging options for a web application. We will focus on the JavaScript-specific panels, but we will cover each panel briefly for anyone who's not so familiar with them.

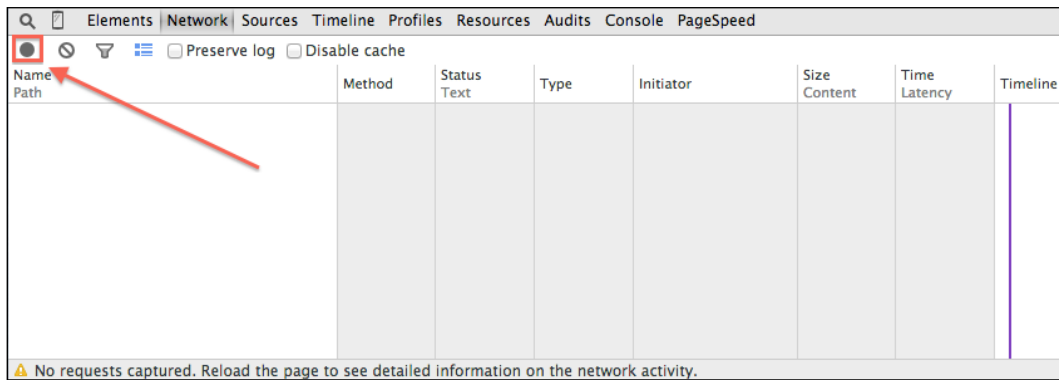
The Elements panel

The **Elements** panel displays both the HTML page's source code and DOM Explorer, allowing developers to inspect changes in the DOM. We can highlight elements by either placing the mouse over the DOM tree, or by using the magnifying lens as indicated in the following screenshot:



The Network panel

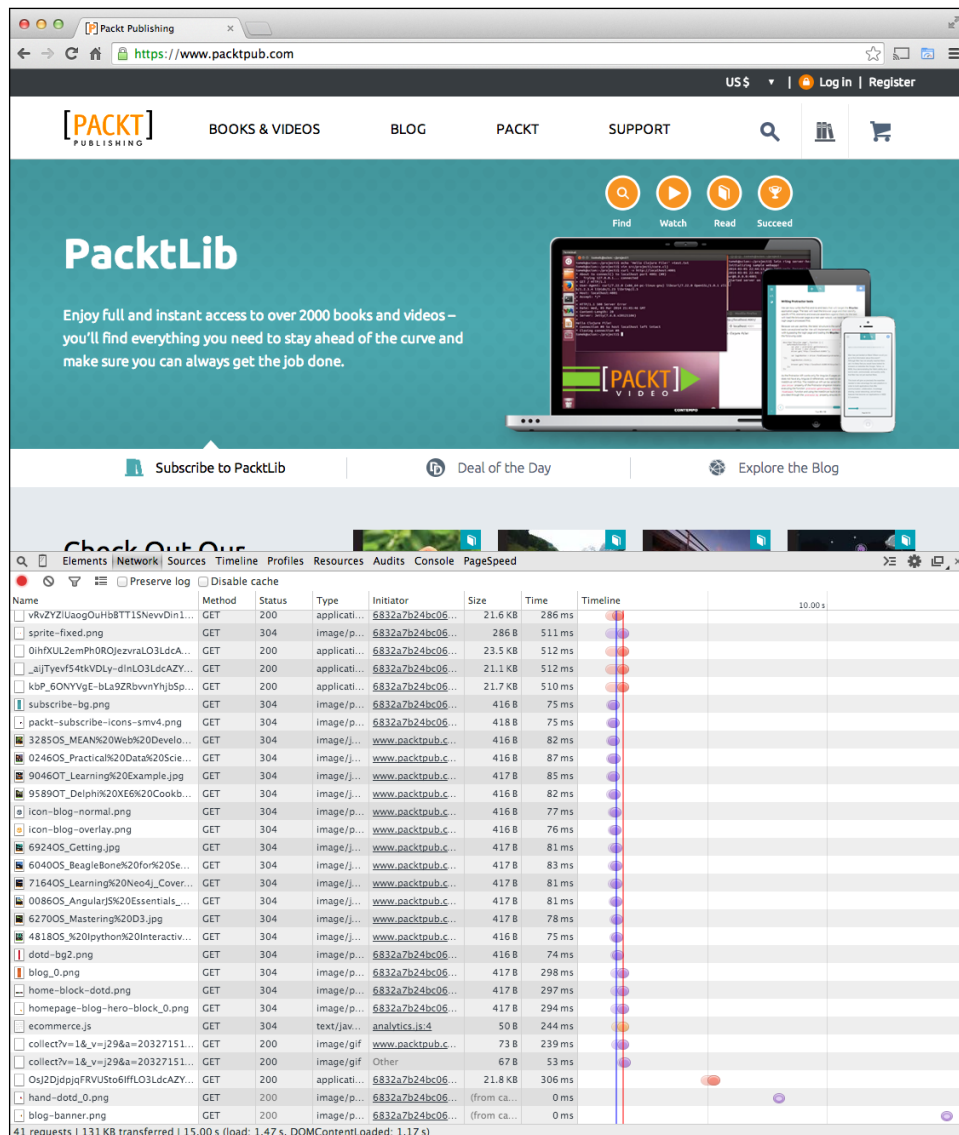
The **Network** panel displays page download speeds of all the resources and the code it contains. Let's test this out by going to <http://www.packtpub.com/> and opening the **Network** panel (located right next to **Elements**). Click the record button on the top left of the panels as shown in the following screenshot:



Now, let's refresh the page with the record button on. We can see which page resources are taking longer to load in our web page. This is important when considering loading resources in JavaScript. If we target an element or a script that doesn't yet exist in our DOM, an error could occur.

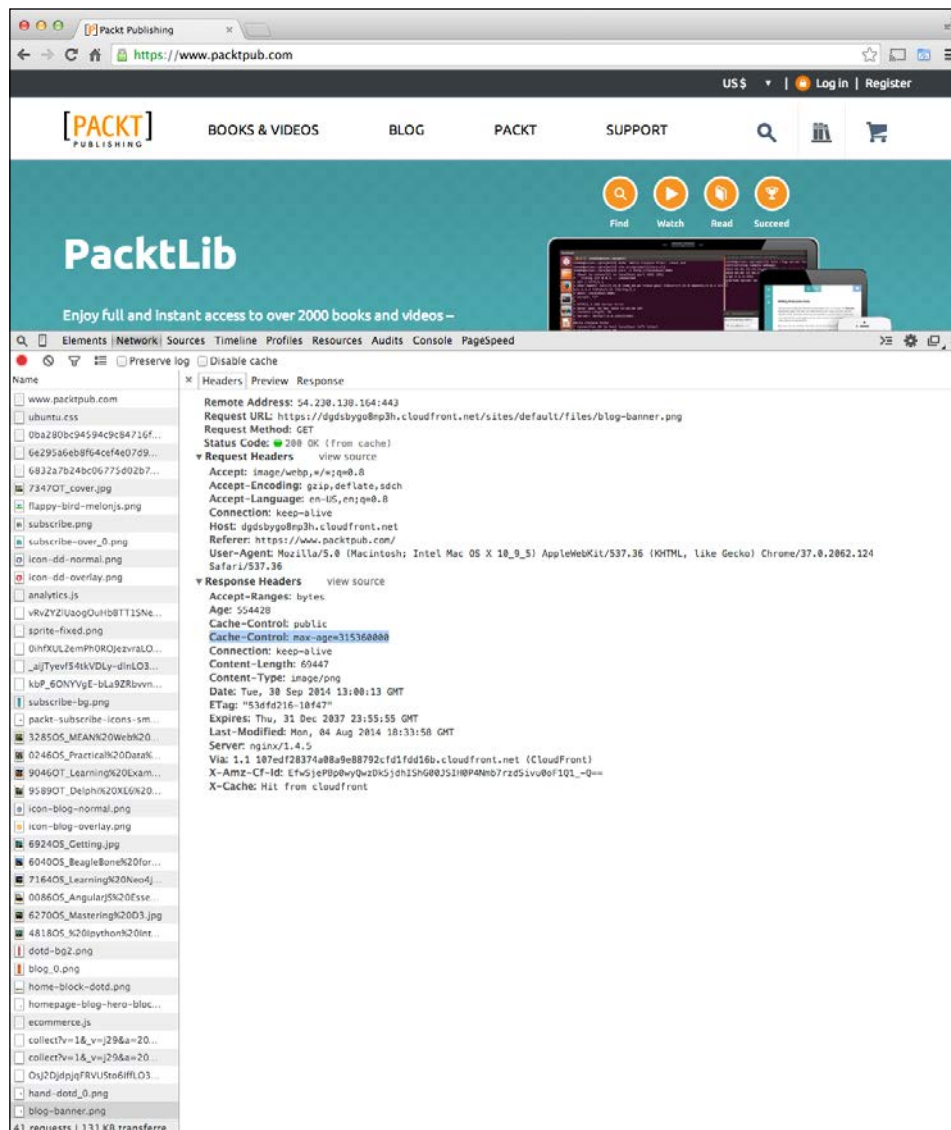
Detecting Performance

If we take a look at the following screen, we can see that the `blog-banner.png` graphic is taking the longest time to load on `http://www.packtpub.com/`.



We can also select a resource as well; let's click on one of the image resources. (I'll choose `blog-banner.png`, this may or may not exist on your page. If you are testing on first load, give the site a few moments to load). When we select it, we can see a new sub-panel appear showing a preview of the image if it's a graphic or the source code if it's a JavaScript or JSON file.

We also have tabs in the subpanel, one of which is called **Response**. This gives information to POST event resources found by DevTools. We also have a tab called **Headers**. The **Headers** tab displays request information for that file, including (more importantly) whether the image uses any server-side caching. In this case, our `blog-banner.png` file has a `Cache-control: max-age` value indicating a maximum cache age of 3153600000 seconds or ten years. We can also see the full Request URL noting that it's using a `cloudfront.net` URL, so we can infer that the image is using Amazon S3 for caching and distribution, as shown for both in the following screenshot:

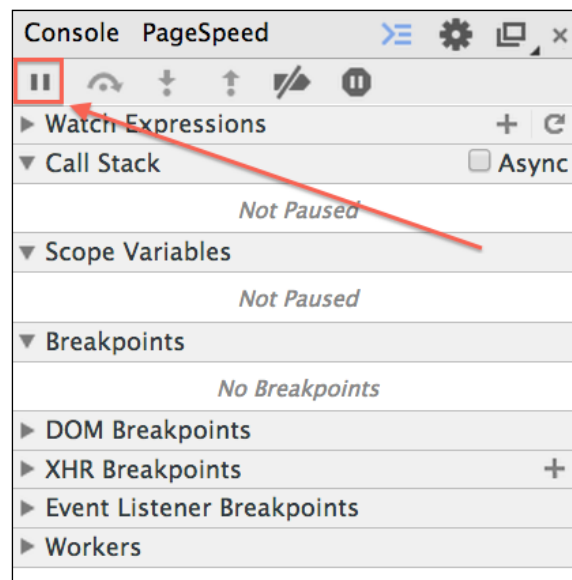


The Sources panel

Here we are going to learn about the **Sources** panel, with the help of the following aspects:


Debugger basic usage

The **Sources** panel is home to most JavaScript developers; it's where we debug our JavaScript applications. Using it is pretty simple; click the pause button up on the top left section right near the **Watch Expressions** option, as shown in the following screenshot:



Testing the debugger

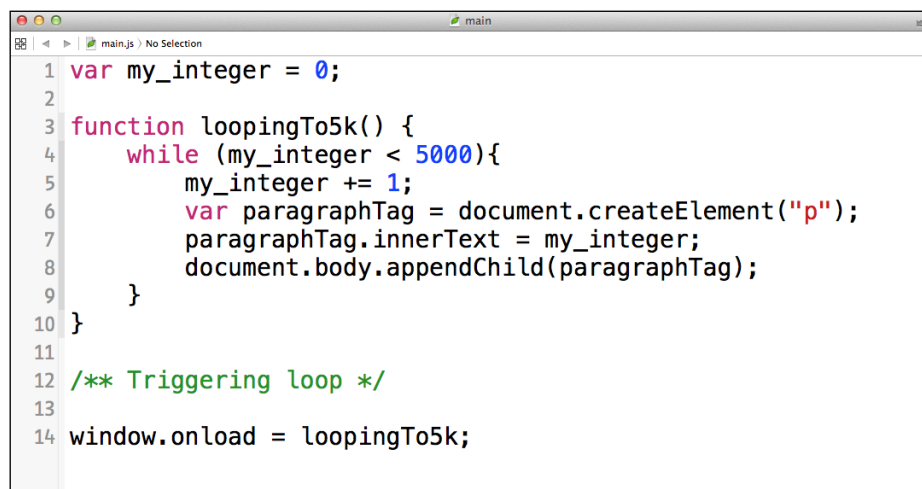
Let's try out the debugger. Open the 01 folder inside the Chapter_4 folder, in our Exercise_Files folder in the code bundle provided by Packt Publishing's website. Inside it, we can see a very simple code sample, and we also have an HTML5 index.html page, which looks like the following screenshot in our source view:



```
1 <!DOCTYPE HTML>
2 <html>
3 <head>
4   <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
5   <title>Chapter 3 - Code Example 01</title>
6   <style type="text/css">
7     body {
8       font-family: sans-serif;
9       font-size: 2em;
10    }
11  </style>
12 </head>
13 <body>
14
15
16 <script src="main.js"></script>
17 </body>
18 </html>
```

We can see that we have a very empty web page with some styling added for the body tag; we've also added a `main.js` external JavaScript file handling all of our page logic. What we are going to do here is inspect a function with a `while` loop inside.

The loop will append the `document.body` tag with the `paragraphTag` variable, each with an index variable called as a global variable named `my_integer` outside the `while` loop, which is contained in a `loopingTo5k()` function. This is called on line 14, where it is being triggered by a `window.onload` event, as shown in the next screenshot displaying the `main.js` source view:

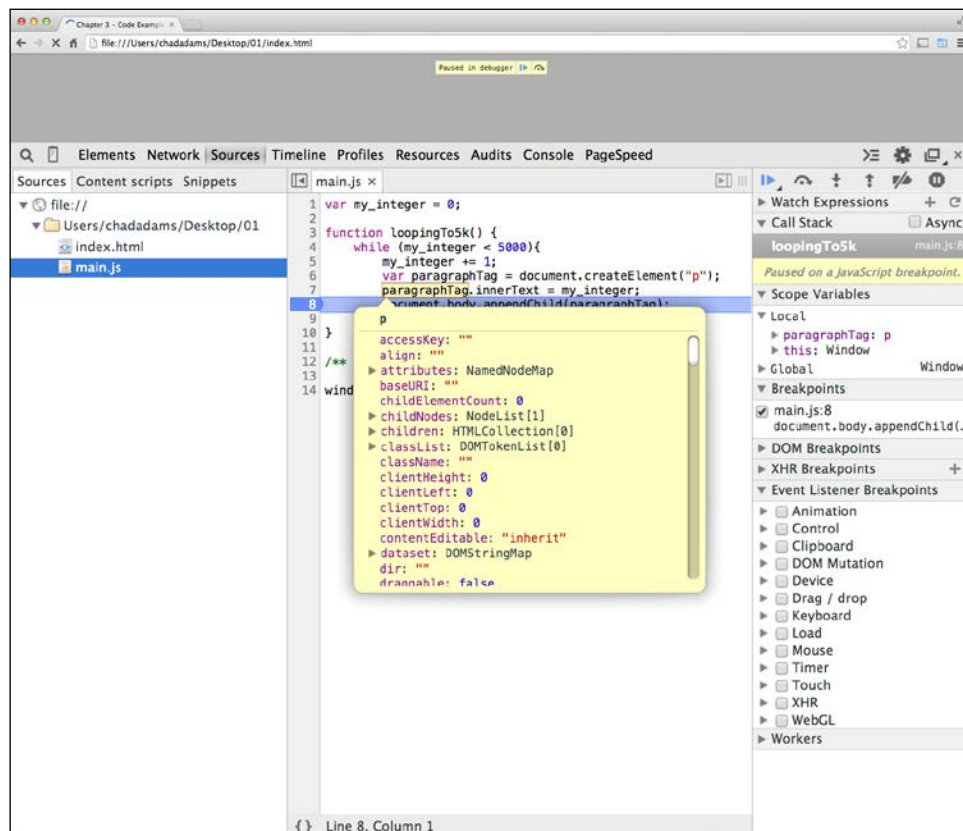


```
1 var my_integer = 0;
2
3 function loopingTo5k() {
4   while (my_integer < 5000){
5     my_integer += 1;
6     var paragraphTag = document.createElement("p");
7     paragraphTag.innerText = my_integer;
8     document.body.appendChild(paragraphTag);
9   }
10 }
11
12 /** Triggering loop */
13
14 window.onload = loopingTo5k;
```

With our source code in place, let's go ahead and run our page in Chrome with our **Sources** panel open. If we look at the screen, we can see a set of numbers moving down the page in a sequential order ending at **5000** on the last line of the document.

Let's select the `main.js` file in our **Sources** panel, add a breakpoint to line 8 of our source code, and see what the **Sources** panel can do. Now with our breakpoint set, let's refresh the page. When we do this, we can see the page graying out with a note in yellow at the top indicating that we are paused in our debugger, and line #8 in our `main.js` file is highlighted in blue, noting where the debugger paused.

We can also see the **Scope Variables** option, which shows all the properties and objects of a given scope at the time of execution; in this case, the scope is inside the `loopingTo5k()` function. To get more information, we can refer to the right section of the **Sources** panel and look at the Local tree for information, or we can mouse over the objects in our code file for more information. As shown in the following screenshot, I've highlighted the `document.body` object in my function's scope, creating a new paragraph object in JavaScript.



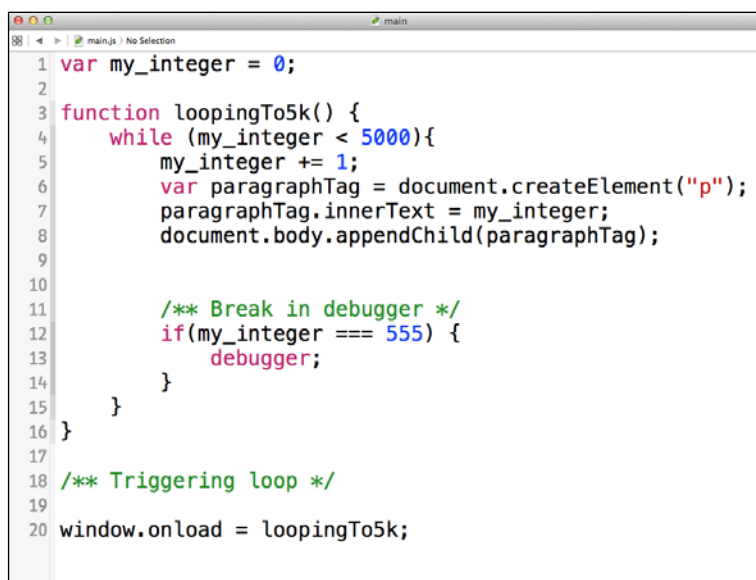
When we're finished with debugging, we can press the play button highlighted blue in the **Sources** panel, or we can **Step Over** our function via the control next to the play button and move on to our next function. Keep in mind that, if we have any further breakpoints, they will break further down the source file in our web page. To remove breakpoints, we can drag them off our line number column and press play to resume without debugging.

Using the debugger keyword

A little known feature in JavaScript programming is the **debugger** keyword; it's a very simple helper function. When running code, it will trigger the **Sources** panel or another JavaScript debugger connected to break automatically; this is helpful when going over large code bases or having trouble breaking on a certain line.

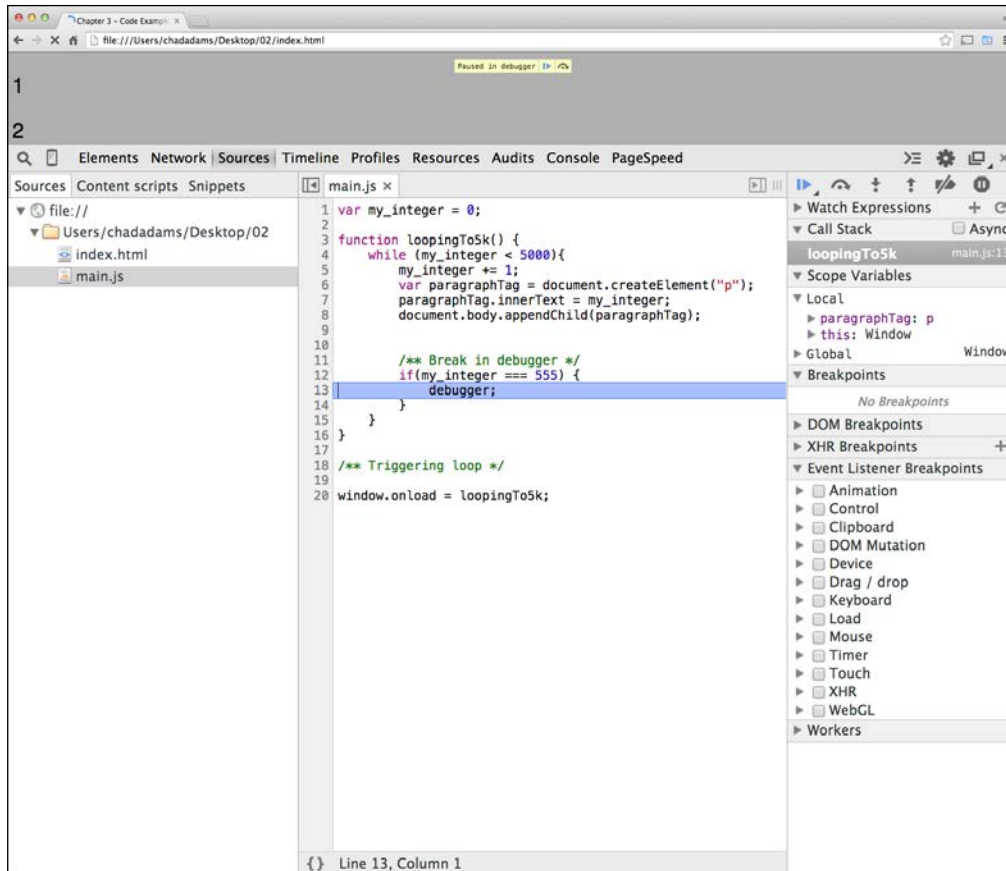
Let's say that, in our example code thus far, we had a while loop, causing an issue with our code at the 555 iteration of `my_integer`. If we had to step through this, it would take 555 presses of the play button to get there. However, there is a way around this.

To demonstrate this, I've set up a copy of these source files and saved them in the 02 folder in the code bundle provided to you through Packt Publishing's website under the Chapter_03 folder in the Exercise_Files folder. I've only made one change here in the code: adding a conditional `if` statement on lines 12 through 14, ensuring `my_integer` is equal to 555. If that is applied, I would call the debugger by simply writing `debugger` with a semicolon to end the line, as shown in the following screenshot:

A screenshot of a web browser's developer tools 'Sources' panel. The code editor shows a JavaScript file named 'main.js'. The code includes a variable 'my_integer' set to 0, a function 'loopingTo5k()' with a 'while' loop that increments 'my_integer' and appends paragraph tags to the document body. A conditional 'if' statement is added on line 12, which calls 'debugger;' when 'my_integer' equals 555. The code is as follows:

```
1 var my_integer = 0;
2
3 function loopingTo5k() {
4   while (my_integer < 5000){
5     my_integer += 1;
6     var paragraphTag = document.createElement("p");
7     paragraphTag.innerText = my_integer;
8     document.body.appendChild(paragraphTag);
9
10
11     /** Break in debugger */
12     if(my_integer === 555) {
13       debugger;
14     }
15   }
16 }
17
18 /** Triggering loop */
19
20 window.onload = loopingTo5k;
```

Now calling debugger is easy. Let's load our `index.html` file again with our debugger code, and here we can see that, without setting a breakpoint, our **Sources** panel automatically detects the line and sets the breakpoint without iterating through each loop (as shown in the following screenshot):



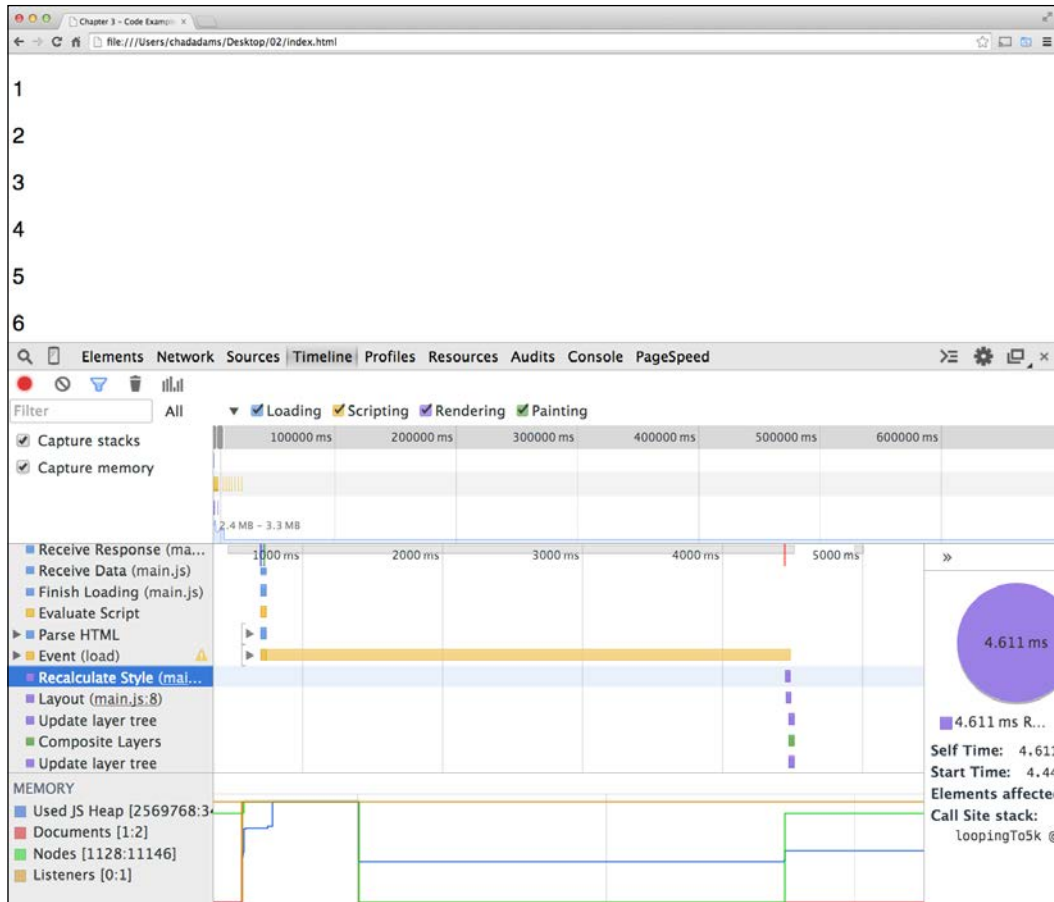
The Timeline panel

Here, we are going to learn about the **Timeline** panel with the help of the following aspects:

Using the Timeline panel

The **Timeline** panel allows us to detect the overall web page performance with respect to JavaScript; it also allows us to inspect browser rendering events. To use the **Timeline** panel, all we need to do is click the record button and reload the page in Chrome.

In the **Timeline** inspector, there are four types of events that the **Timeline** panel shows. These are **Loading**, **Scripting**, **Rendering**, and **Painting** events. I've loaded the example file (02), discussed in an earlier section, showing the events running through the **Timeline** panel, as seen in the following screenshot:



The Loading event

The **Loading** event handles requests and responses; typically these are loading external scripts and files as well as POST requests for data leaving the page. Loading events also include the initial parsing of HTML code. In Google Chrome's **Timeline**, these show up in blue.

The Scripting event

The **Scripting** event occurs when the browser reads and interprets JavaScript code. In the **Timeline** panel, you can expand a **Scripting** event and see at what point a function was received in the browser. **Scripting** events appear as yellow lines in Google Chrome.

The Rendering event

The **Rendering** event occurs when image files and scripts affect the DOM; this can be when an image is loaded without a size specified in an `image` tag, or if a JavaScript file updates the CSS of a page after the page is loaded.

The Painting event

The **Painting** event is the last type of events and typically is used in updating the UI. Unlike **Rendering** event, the **Painting** event occurs when the browser redraws an image on the screen. For desktop JavaScript development, **Painting** events aren't usually a concern, but become strongly concerning when we start looking at mobile web browsers.

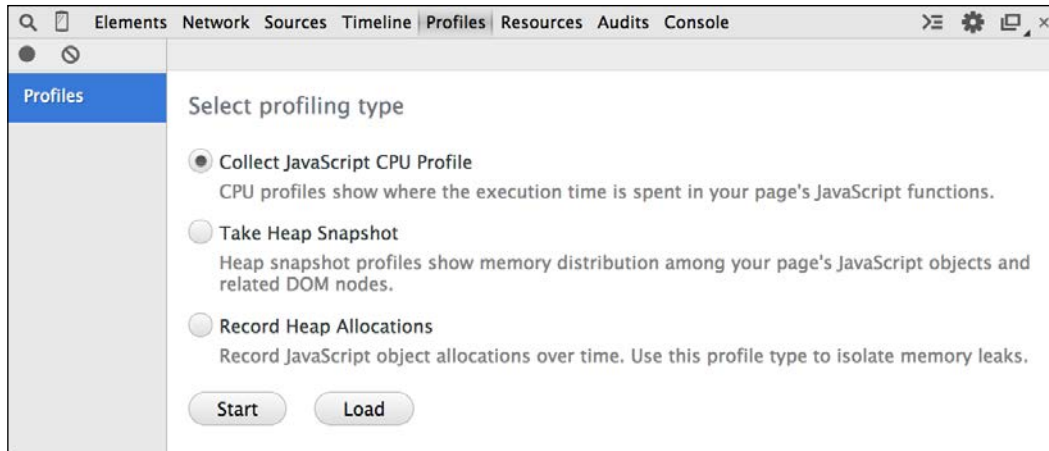
Typically the **Painting** event is forced when an element's display is updated from its original. They can also be triggered by updates to an element, such as an element's `top` or `left` positioning.

The Profile panel

The **Profile** panel helps a developer analyze a web page's CPU profile and take heap snapshots of the JavaScript used. A CPU profile snapshot is helpful when it comes to checking large complex applications to see what files may cause issues in terms of object size.

A JavaScript heap snapshot is a compiled list of objects found in the page's overall JavaScript. This includes not only the code written by us, but also the code built into the browser, such as the document or console objects, giving an overall list of all possible objects in an application.

Using the **Profile** panel is similar to the **Timeline** panel; select either the **Take Heap Snapshot** or the **Collect JavaScript CPU Profile** option, and then click **Start**, followed by reloading the page. In the following screenshot, I have selected the **Collect JavaScript CPU Profile** option:

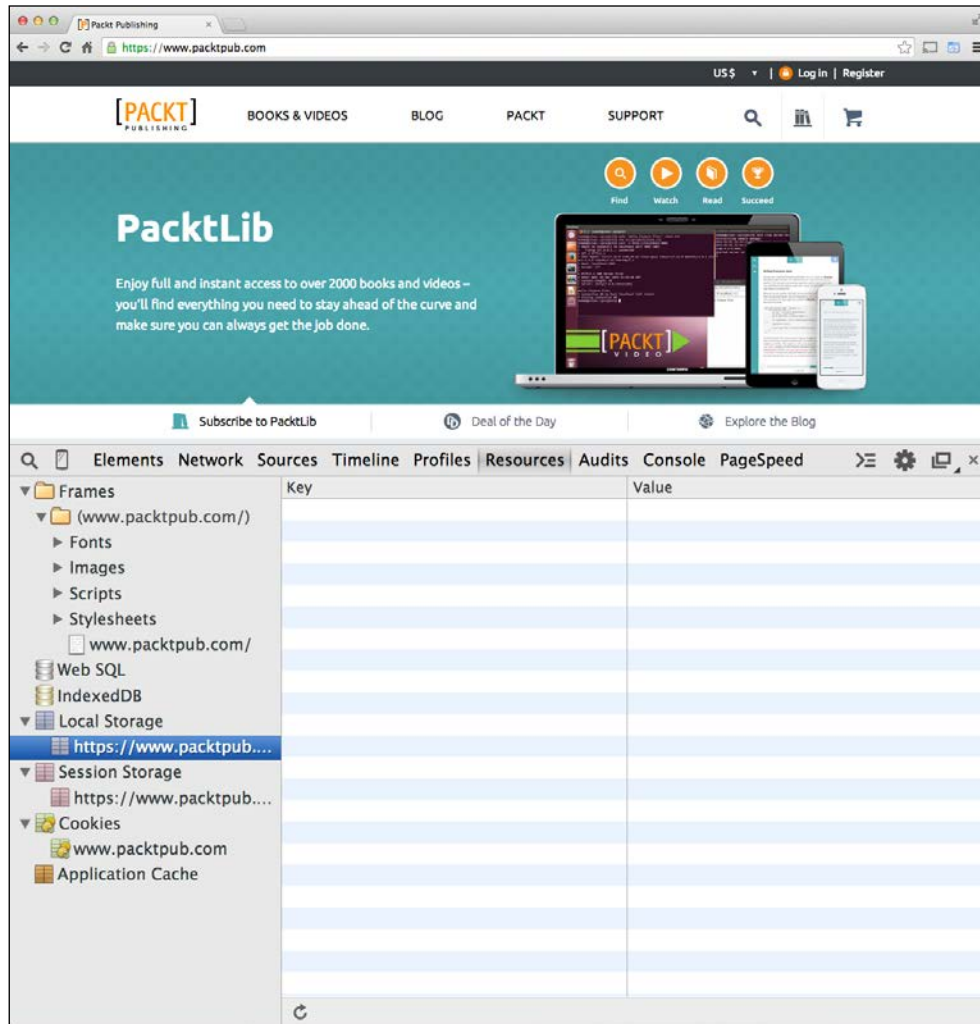


The Resources panel

The **Resources** panel lists all files associated with the web page being viewed in the **Developer tools** option, each of which can be sorted by the type of file; developers can individually view each file. It also shows images on the page along with their information such as **Dimensions**, **File size**, **MIME type**, and source **URL**.

More importantly, the **Resources** panel is home to any browser data storage, which includes **Web SQL**, **IndexedDB**, **Local Storage**, **Session Storage**, and **Cookies**. Users can look at a page's **Key-Value** pair values in the browser's storage data. This is helpful in testing storage state and store values in JavaScript code.

Viewing the Key-Value pairs is easy; in the **Resources** panel, select the storage type and take a look at the key values table, as shown in the following screenshot using Packt Publishing's website while viewing local storage:

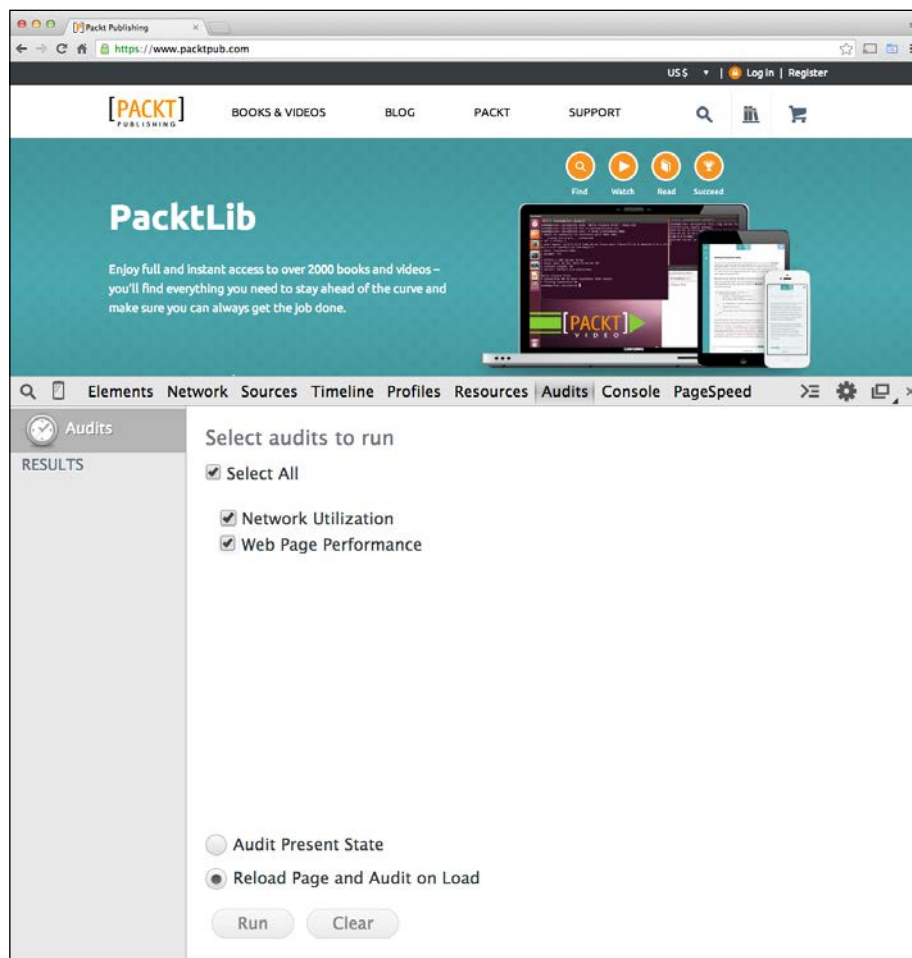


The Audits panel

Here, we are going to learn about the **Audits** panel, with the help of the following aspects.

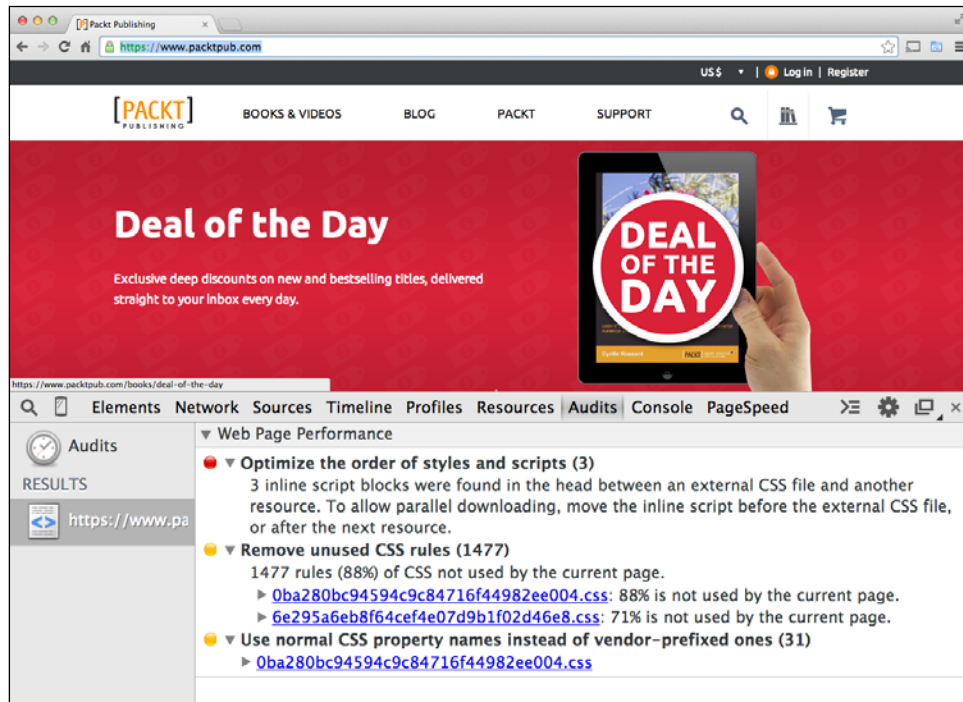
Interacting with the Audits panel

The **Audits** panel *audits* the full web page's application **Network Utilization** and overall **Web Page Performance**; this is one of the easier to use and more straightforward panels in the **Developer tools** options provided by the browser. Using the **Audits** panel is easy as well. First, open up Packt Publishing's website again, select the **Audits** panel using the **Developer tools** option, and then check the **Select All** option; this will test network speeds and the overall web page performance. Lastly, be sure to set the radio button **Reload Page and Audit on Load**, prior to clicking the **Run** button. This will ensure that the audit test checks for network usage properly rather than a cached state, just as its shown in the following screenshot:



Getting Suggestions for JavaScript quality

If we're only checking for JavaScript performance, uncheck the **Network Utilization** option and run the test as well; we need to keep this in mind if we're testing for a specific point in our application. We will need to switch the radio button to **Audit Present State**, and click **Run** to get suggestions for the current state of the web application. Let's run the test on `https://www.packtpub.com/`, and then select the file under **Results**. Let's take a look at the performance improvement suggestions as shown in the following screenshot:



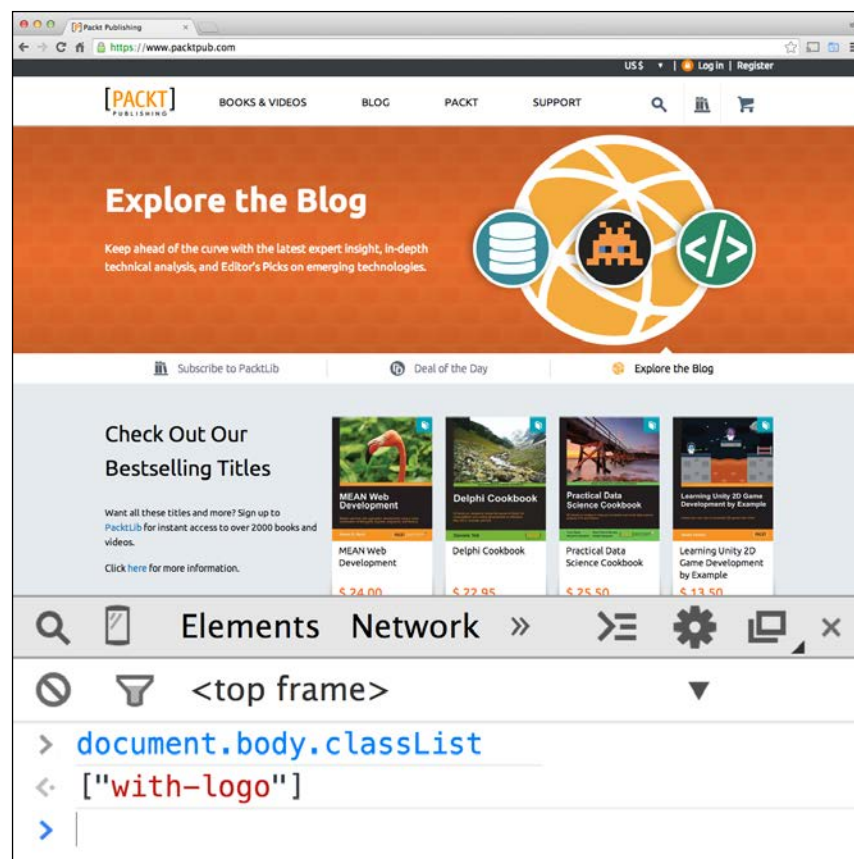
If we look closely, we can see very readable suggestions, with respect to our page's JavaScript code, affecting the overall page performance. In this case, the audit has detected 3 inline scripts and recommends moving the inline scripts to improve performances. There is also feedback on how many CSS rules included in the page are not used (on this page at least). It also tells us whether vendor prefixes are being used in CSS and not web standard properties. All of these suggestions are very helpful.

The Console panel

The last out-of-the-box panel is the **Console** panel. It's the simplest panel here, but it's also where developers on JavaScript spend most of their time. Now my assumption is that we're fairly familiar with the **Console** panel at this point, so I won't deep dive into this panel too much. We can test code in the console and search for objects, DOM elements, and attributes in a page. For instance, let's say I type the following into the console while on Packt Publishing's website:

```
document.body.classList
```

This should return a JavaScript array on the next line showing all the classes available to us, and it does show one having `with-logo` as the class name, as shown in the following screenshot:



The **Console** panel and the **Console** API in Chrome are constantly evolving in terms of features in Chrome's **Developer tools**. To keep up with some of the newer tools, check out the Chrome's DevTools Console API page available at <https://developer.chrome.com/devtools/docs/console>, which shows how to use the console for custom outputs such as `console.table()` and `console.profile()` to make developing in the console much easier.

Summary

In this chapter, we explored the base panels that come with the consumer version of the Google Chrome **Developer tools**; many of these tools carry over to other inspectors and developer tools (this was also covered earlier in the chapter). I encourage you to read up on each and see where and how the code is inspected in other inspectors as well as in Chrome's **Developer tools**.

In the next chapter, we'll get into JavaScript performance coding without any help.

5

Operators, Loops, and Timers

In previous chapters, we reviewed the basic tools used in JavaScript development. We looked at IDEs, code editors, and *JSLint*, a JavaScript code validator that not only showed us where our code contained issues, but it also gave us warnings and suggestions on how to improve our code.

We also learned about the `console.time` and `console.timeEnd` methods that allowed us to quickly test our code execution performance. Finally, we learned about creating a basic build system to ensure our final code base is optimized and bug-free.

It's important to say that all of these tools and techniques are essential to write high-performance code, not because of the JavaScript you know, but because of the JavaScript you don't know. JavaScript is a language that anyone can pick up and start writing code without knowing object-oriented programming or knowing a pattern such as **Model View Controller (MVC)**; over the years, however it's been modified to accommodate these higher-level programming concepts (hacked or otherwise).

The flip side of an easy-to-use language is that it's very easy to write bugs or even nonoptimized code; this effect is doubled and even tripled if we're writing complex JavaScript. As mentioned in past chapters, one characteristic of JavaScript developers in general is that *we are human, and we make mistakes*. Much of this is just a lack of awareness on the part of the developer, which is why using build systems and code checkers such as *JSLint* is so important, long before we write perfect high-performance JavaScript; if we don't, these tools have us covered.

In this chapter, we are leaving tools and build systems behind and are getting into JavaScript performance concepts head-on, breaking up the subject matter across two chapters, starting with following topics:

- Operators
- Loops
- Timers

Operators

In this section, we are going to learn efficient ways to create `for` loops using the comparison operator.

The comparison operator

The comparison operator, `==`, is an operator that's common in JavaScript development (typically in `if` statements); it equates one object with another and returns a boolean, (`true` or `false`) value. It's pretty straightforward and is very common in C-based languages.

Because of this, it's easy to take advantage of this operator and use it across a large code base. The reality of this is that the equals operator is slow compared to using the `===` strict comparison operator, which also compares object types as well as the object's value. As the JavaScript interpreter doesn't have to confirm the type before checking the equality, it operates faster than using the double equals operator.

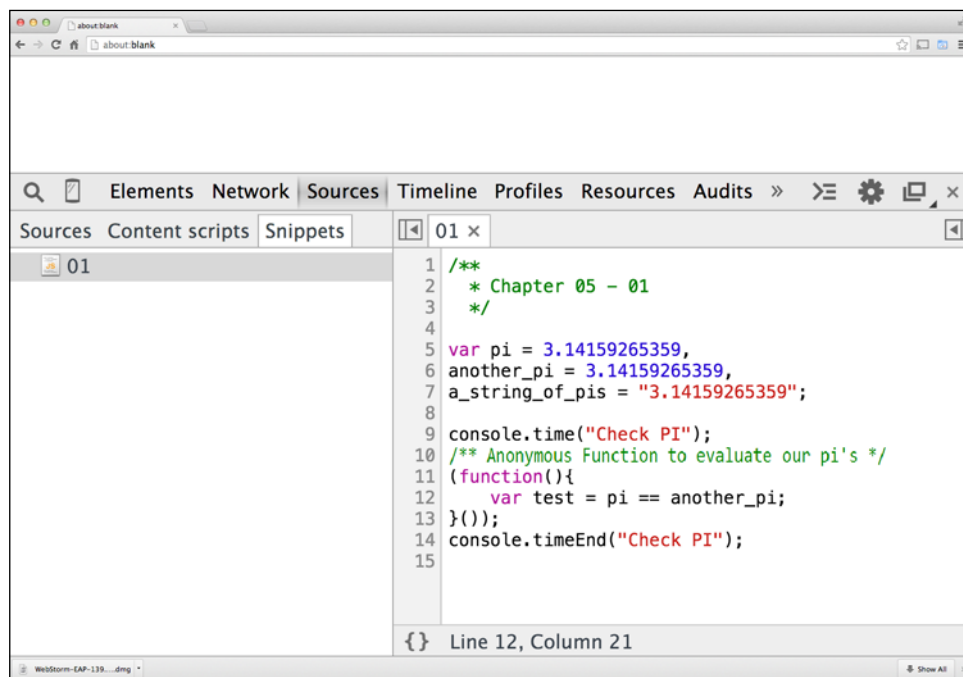
Is strict faster?

Let's test this using the `console.time` method. In the following screenshot we have a `05_01.js` code sample; we can also see this sample in the example files of this book, provided on the Packt Publishing's website:

```
1  /**
2   * Chapter 05 - 01
3   */
4
5  var pi = 3.14159265359,
6      another_pi = 3.14159265359,
7      a_string_of_pis = "3.14159265359";
8
9  console.time("Check PI");
10 /** Anonymous Function to evaluate our pi's */
11 (function () {
12     var test = pi == another_pi;
13 }());
14 console.timeEnd("Check PI");
```

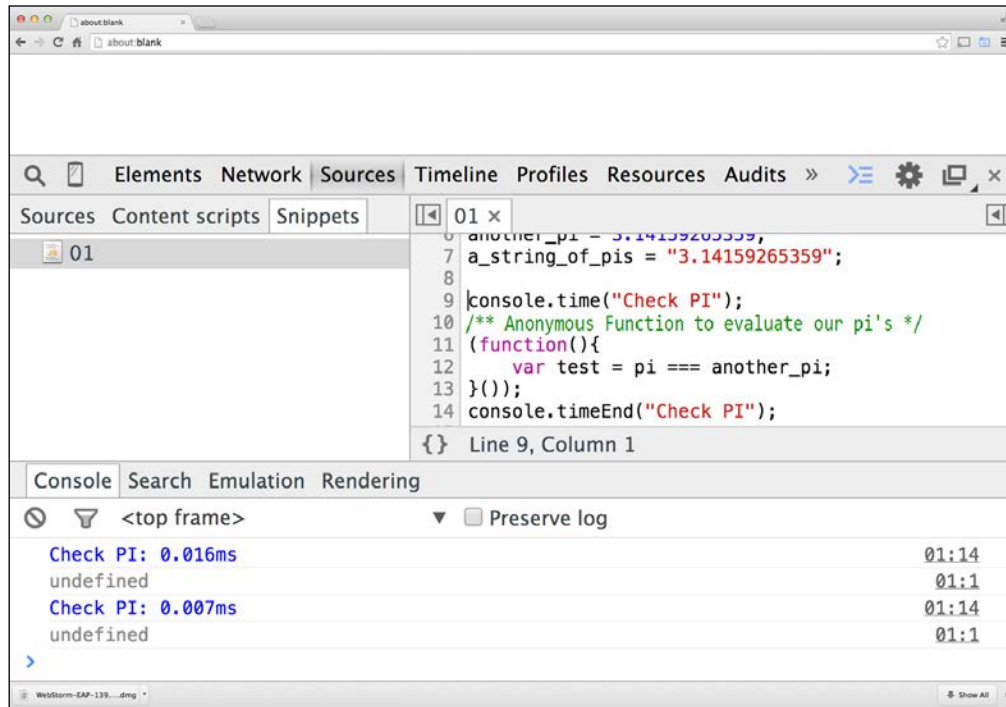
Here, we have three variables on lines 5, 6, and 7; two of these variables are floats referencing the pi value, and the last variable is a string with the same pi value. We then have an anonymous function with a test variable on line number 12, which equates both our floats with a double equals operator. Wrapped around the function, we have `console.time` and `console.timeEnd` functions on lines 9 and 14, respectively.

Let's run this in the Chrome browser; open up Chrome followed by **Developer tools** from the **More tools** option on an `about:blank` tab, and then open the **Snippets** tab in the right-hand side column under the **Sources** panel. The **Snippets** tab is like a scratchboard built to test JavaScript code; right-click in the tab content area, and select **New**. Save your snippet with a name and copy the code from the example, as shown in the following screenshot:



Next, right-click on the code snippet in the left-hand side rail and click on **Run**. You'll notice the console appear at the bottom of the **Developer tools** window. We can also see a `Check PI: 0.016ms` console message. This shows us that running the comparison operator on this simple evaluation takes 0.016 ms to complete. What if we changed the comparison operator to a strict comparison operator to see what the result would be?

On changing the operator, we can see that our second `console.time` message is `Check PI: 0.007ms`. This is a simple example, sure, but it proves that the code runs faster using strict type checking with strict comparison operators.



Loops

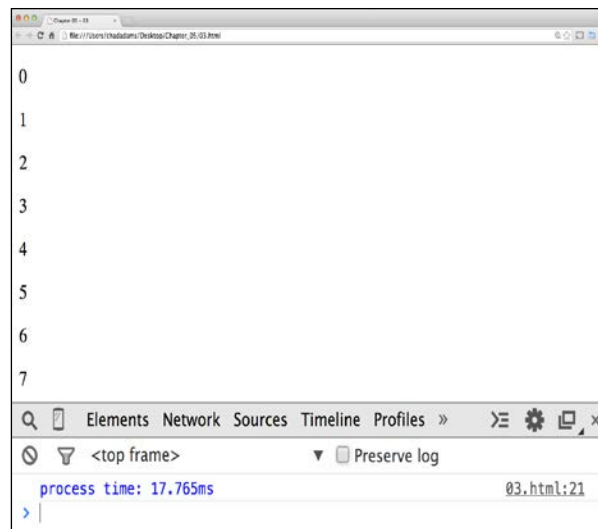
In this section, we are going to learn efficient ways to create `for` loops in detail.

How loops affect performance

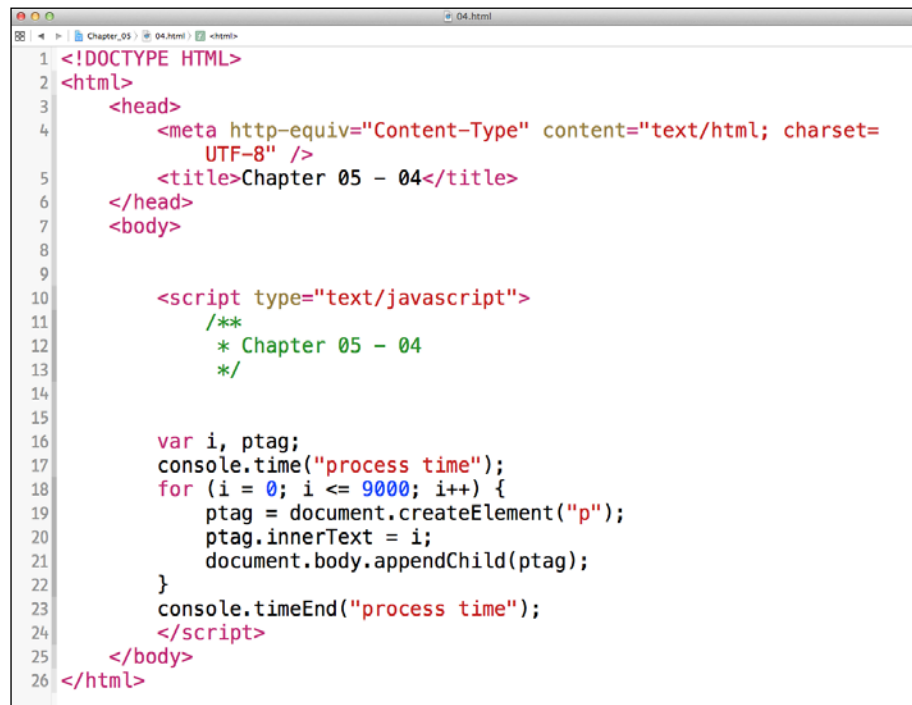
Loops are a very common way to iterate through large blobs of data or objects and iterating through each instance of a DOM object or a piece of data. Let's say we have a simple loop that generates a `p` paragraph tag and appends the page with an inner text value of the `i` integer in the loop, with a maximum limit of 9000. Let's take a look at the following code sample and see how this is done. I've created a simple HTML5 page with a `script` tag that includes the code on line 10, as shown next:

```
1 <!DOCTYPE HTML>
2 <html>
3   <head>
4     <meta http-equiv="Content-Type" content="text/html; charset=
      UTF-8" />
5     <title>Chapter 05 - 03</title>
6   </head>
7   <body>
8
9
10  <script type="text/javascript">
11    /**
12     * Chapter 05 - 03
13     */
14
15    console.time("process time");
16    for (var i = 0; i <= 9000; i++) {
17      var ptag = document.createElement("p");
18      ptag.innerText = i;
19      document.body.appendChild(ptag);
20    }
21    console.timeEnd("process time");
22  </script>
23 </body>
24 </html>
```

So, how is this code process-intensive? For starters, if we look at line number 17, we can see a variable called `ptag` that was created to create a blank paragraph tag in our DOM. We then apply the integer's current value in the loop to the `innerText` property of the `ptag` variable; and lastly, we apply the newly created paragraph tag into the DOM with the value we specified at that point in the loop. For performance testing, we also wrapped the `for` loop in a `console.time` wrapper method to check the performance speed. If we run this in Chrome, we should get a page with a line for each number created in the `for` loop along with a `console.time` method with a `process time` label, as shown in the following screenshot:

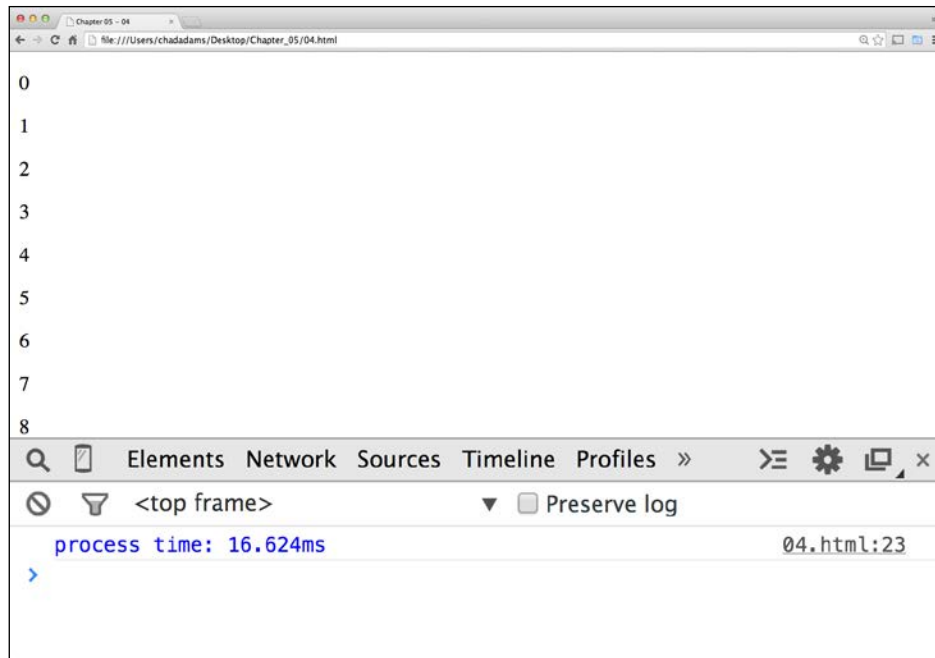


Looking at our `process time` label, we can see that processing this block of code about takes 18 milliseconds to complete. This is not great, but we can make it better; let's update our code and move the `ptag` variable and our `i` integer variable outside the `for` loop so that they don't get recreated with each iteration of the `for` loop. Let's see what this looks like by updating our code, as in the following screenshot:



```
1 <!DOCTYPE HTML>
2 <html>
3   <head>
4     <meta http-equiv="Content-Type" content="text/html; charset=
      UTF-8" />
5     <title>Chapter 05 - 04</title>
6   </head>
7   <body>
8
9
10
11     <script type="text/javascript">
12       /**
13        * Chapter 05 - 04
14        */
15
16       var i, ptag;
17       console.time("process time");
18       for (i = 0; i <= 9000; i++) {
19         ptag = document.createElement("p");
20         ptag.innerText = i;
21         document.body.appendChild(ptag);
22       }
23       console.timeEnd("process time");
24     </script>
25   </body>
26 </html>
```

Notice that, on line number 16, we've moved the `i` and `ptag` variables outside the loop, and we are reassigning values and objects created in the loop rather than creating a unique scope for each loop pass. If we rerun our page, we should see the same body tag get updated with a slightly smaller performance number than before; in the following case, it should run in the range 15–17 milliseconds:



The reverse loop performance myth

A new idea that seems to have appeared in JavaScript developer circles is the concept of a reverse `for` loop. A reverse `for` loop is written just like a loop, but the loop counts backward rather than forward.

The idea behind a reverse loop is that, by counting backward, some JavaScript interpreters run loops faster. Let's test this and see whether this actually improves the speed of a `for` loop. First, let's create a `for` loop, counting forward from 9000; we won't include any logic in the `for` loop, with the exception of adding an external variable called `result`.

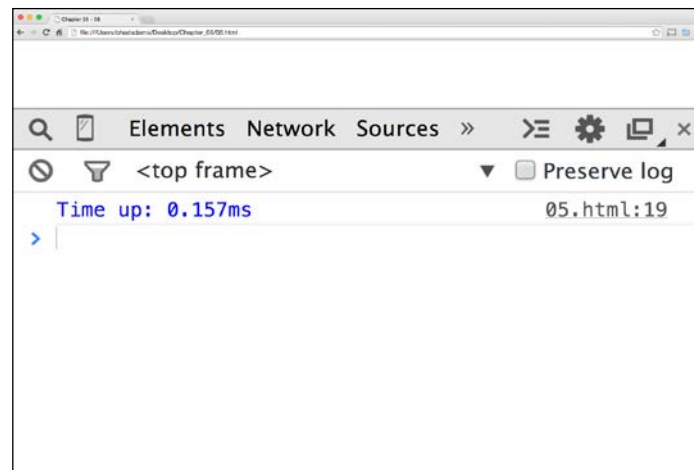
Using our `result` variable with an increment, we can determine whether we are counting as we should and triggering a line of code at the end of 9000 in both a *reverse* loop and a standard `for` loop. In our case, a `console.timeEnd` function, as shown in the following code, is in its own HTML page, with a script tag at the bottom.



```
1 <!DOCTYPE HTML>
2 <html>
3   <head>
4     <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
5     <title>Chapter 05 - 05</title>
6   </head>
7   <body>
8
9     <script type="text/javascript">
10      /**
11       * Chapter 05 - 05
12       */
13      var result = 0;
14      console.time("Time up");
15      for (var i = 0; i <= 9000; i++) {
16        result++;
17        /** Trigger a timeEnd, when the loop hit's 9000. */
18        if (result === 9000) {
19          console.timeEnd("Time up");
20        }
21      }
22    </script>
23  </body>
24 </html>
```

Let's take a look at the code sample. On line 13, we can see that we declare our `result` variable before starting our `for` loop while, on line number 14, we start the `console.time` wrapper method that has a label called `Time up`. On line 15, we start our `for` loop and increment `result` on line 16. Finally, on line 18, we have a condition where we ask whether the result is equal to 9000, and we execute our `timeEnd` function on line 19.

If we load the page with our `for` loop script inside the `body` tag, our console in **Developer tools** should output the following information:



So, our `console.time` object tells us that our standard `for` loop with a maximum value of 9000 takes roughly 0.15 milliseconds to process in Google Chrome. Having nothing else included on the HTML page, which isn't hosted on a server, ensures that network lag isn't a factor. This is a good baseline with which we can compare a reverse loop.

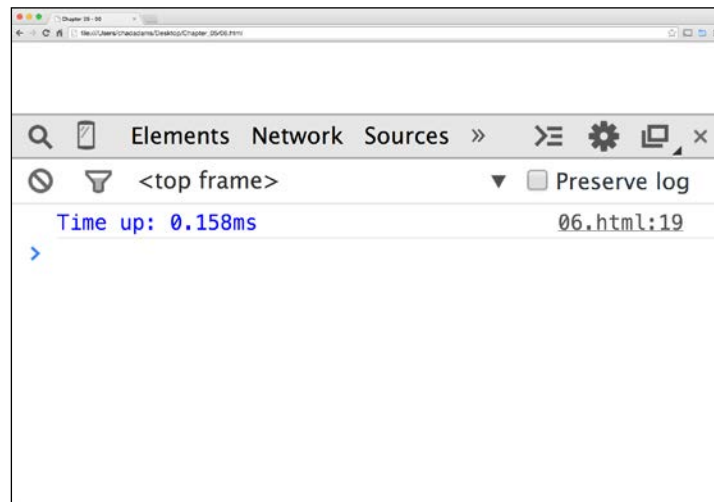
Now, let's test a reverse `for` loop; here, we've created an updated version of the `for` loop, including our `result` variable. This is similar to the previous process, but let's take a look at the code sample in the next screenshot:

```
1 <!DOCTYPE HTML>
2 <html>
3   <head>
4     <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
5     <title>Chapter 05 - 06</title>
6   </head>
7   <body>
8
9     <script type="text/javascript">
10      /**
11       * Chapter 05 - 06
12       */
13      var result = 0;
14      console.time("Time up");
15      for (var i = 9000; i > 0; i--) {
16        /** Trigger a timeEnd, when the loop hit's 0. */
17        result++;
18        if (result === 9000) {
19          console.timeEnd("Time up");
20        }
21      }
22    </script>
23  </body>
24 </html>
```

If we look at line number 15 in this code sample, we can see that we've altered this line a bit so that the loop counts backward rather than forward. We start by setting the increment variable, `i` in this case, with a value of 9000, and then we test whether `i` is greater than 0. If it is, we decrease the `i` value by one.

On line 17, we still increment our `result` variable as we did previously. This way, rather than using the `for` loop's decrement variable `i`, the `result` variable exists as our count outside the loop, counting up. This is called by the *reverse* loop. When the `result` equals 9000 on line 18, then on line 19 the `console.timeEnd` function is executed.

Let's test this in the **Developer tools** option in our Chrome browser and see what value we get, as shown here:



So, we can see our result in **Developer tools**, and our reverse loop's processing time is around 0.16 milliseconds, which isn't too much of a difference when comparing it with a `for` loop. In many cases, a reverse `for` loop isn't necessary for most JavaScript projects unless we need to count backward for a project.

Timers

Here, we are going to learn about optimizing JavaScript timers in detail.

What are timers and how do they affect performance?

Timers are built-in functions of JavaScript that allow the execution of either inline JavaScript code or permit functions to be called at a specific point of time after, or repeatedly during, the life cycle of a JavaScript application.

Timers are a great tool in a JavaScript developer's toolbelt, but they have their own issues when it comes to performance. Consider the fact that the JavaScript language is single-threaded, which means that every line of code in our application cannot be fired at the exact same time as another piece of code in our application. To get around this, we use a built-in function called `setTimeout`.

The `setTimeout` method takes two parameters to delay a block of code from executing; the first is either the name of a function with our code or a line of JavaScript code by itself, followed by an integer specifying the extent by which we want to delay code execution which is in milliseconds.

On the surface, the `setTimeout` function may seem harmless, but consider this. Let's say we have two functions, both being triggered by one `setTimeout` function each with a `for` loop that prints an incremented value of the `for` loop to the console window. Each function will have a different maximum value and the lower count function will be called slightly after the first larger functions of the `for` loop. Let's take a look at the code sample here:

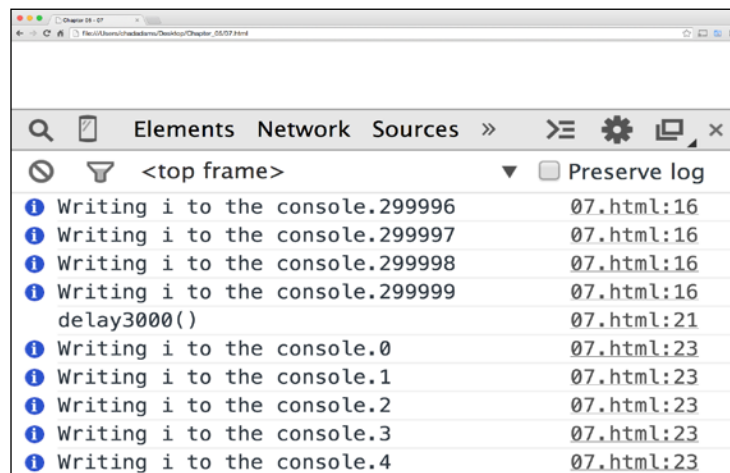
```

1  <!DOCTYPE HTML>
2  <html>
3    <head>
4      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
5      <title>Chapter 05 - 07</title>
6    </head>
7    <body>
8
9      <script type="text/javascript">
10         /**
11          * Chapter 05 - 07
12          */
13         function delay300000() {
14             console.log("delay300000()");
15             for(var i = 0; i < 300000; i++) {
16                 console.info("Writing i to the console." + [i]);
17             }
18         }
19
20         function delay3000() {
21             console.log("delay3000()");
22             for(var i = 0; i < 3000; i++) {
23                 console.info("Writing i to the console." + [i]);
24             }
25         }
26
27         window.onload = function() {
28             setTimeout(delay300000(), 50);
29             setTimeout(delay3000(), 150);
30         }
31     </script>
32 </body>
33 </html>

```


We can see that this is an empty HTML5 page with a script tag with our code on line number 9. On lines 13 and 20, we have the start of two similar functions: one called `delay300000()` and another called `delay3000()`, with each function containing a `for` loop that prints each step of the loop to the console using the `console.info` statement. The `console.info` statement is a type of console print that simply formats the console line to indicate information.

Now, on line 27, we will trigger both functions inside a `window.onload` function, with the larger delay function called 50 milliseconds after the page load and the shorter function called slightly later at 150 milliseconds. Let's try this in Chrome and see what happens in Dev Tools, as shown next:



Here, we can notice quite a bit of lag as we print all these lines to the console. We can also see that we triggered both in a given timeout. In the preceding screenshot, we can see that our `delay3000()` isn't triggered until after our larger function, `delay300000()`, is completed.

Working around single-threading

Sadly, with plain JavaScript, we simply can't "multithread" a function like these two at the same time, but we can incorporate something like a `callback` method in our code. A `callback` method is simply a JavaScript function that's triggered when a function is completed. Let's set up our `delay300000()` function to call back our `delay3000()` method once it's completed. Here's what it would look like:

```

1 <!DOCTYPE HTML>
2 <html>
3   <head>
4     <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
5     <title>Chapter 05 - 08</title>
6   </head>
7   <body>
8
9     <script type="text/javascript">
10      /**
11       * Chapter 05 - 08
12       */
13      function delay300000(callback) {
14        console.log("delay300000()");
15        for(var i = 0; i < 300000; i++) {
16          console.info("Writing i to the console." + [i]);
17        }
18        //Trigger callback after for loop.
19        callback();
20      }
21
22      function Delay3000() {
23        console.log("Delay3000()");
24        for(var i = 0; i < 3000; i++) {
25          console.info("Writing i to the console." + [i]);
26        }
27      }
28
29      window.onload = function() {
30        setTimeout(delay300000(Delay3000), 100);
31      }
32    </script>
33  </body>
34 </html>

```

Looking at our code sample, we can see on line number 13 that we've added a parameter with the name `callback`. It's important to know that here, the naming of our `callback` method isn't important but including a placeholder parameter for a function is. Our placeholder function that will serve as our callback function is `Delay3000()`.

Notice how we renamed `Delay3000` on line 22, capitalizing the *d*. The purpose of this is to indicate to the JavaScript interpreter that this is a **constructor**, a function that requires it to be initialized in memory. This is done by capitalizing the first letter in the function's name. You may recall from *Chapter 2, Increasing Code Performance with JSLint*, if we use a capitalized function name JSLint will return a warning that it "thinks" a constructor is being used even if it's a plain function. To keep our interpreter from second-guessing itself, we want to ensure we are writing our functions and objects as we intended.

Finally, we have updated our `onload` function's logic by removing the extra `setTimeout` for `delay3000`, and we added the newly renamed `Delay3000` (without parentheses) as a parameter inside our `delay3000000()` function in the `setTimeout` function. Let's run this again in our browser, and take a look at our console's output.

If we scroll down near the bottom of the console log (after processing the initial `delay3000000()` function call), we can see that our `Delay3000` log message appears after completing the initial function. Using callbacks is a great way to efficiently manage your application's thread and ensure proper load stacking of a heavy application, allowing you to pass parameters after an initial function is completed.

Closing the loop

Lastly, as we can see in this `callback` method example, it's usually not a great idea to use large hundred thousand scaled loops for performance reasons. Always look for better and more efficient ways to break up large loops and call other functions to help balance out the workload.

Also, I encourage you to check out JavaScript **promises**, an EcmaScript 6 feature. While not quite ready for discussion yet in this book, as at the time of writing, promises are still experimental. I encourage you, dear reader, to follow up and learn about what will be a successor to callbacks in JavaScript when it's finalized. You can learn more about promises on Mozilla's Developer Network site at <https://developer.mozilla.org/en-US/>.

Summary

In this chapter, we learned about conditionals and how efficient strict comparisons help our JavaScript perform better at runtime. We also learned about loops and how to optimize loops, preventing objects that are not required for our code base from being repeated over and over in a `for` loop and thus, keeping our code as efficient as possible.

Lastly, we also learned about timers and single-threading in JavaScript applications and how we can use callbacks to keep our code running as smoothly as possible even when we overload it. Next, we cover arrays and prototype creation performance and find out how best to work with them in JavaScript.

6

Constructors, Prototypes, and Arrays

Now that we are getting comfortable with optimizing JavaScript without a linter or an IDE testing our code for us, it's time to dive into more complex optimization, specifically when it comes to memory and object creation. In this chapter, we're going to take a look at optimizing larger JavaScript code bases using constructors, prototypes, and arrays.

We are planning to cover the following topics in the chapter:

- Building with constructors and instance functions
- Alternate constructor functions using prototypes
- Array performance

Building with constructors and instance functions

Here, we will learn about building with constructors and instance functions in the following ways:

A quick word

Depending on skill level, some of us following this book may or may not know exactly what prototypes are in JavaScript. If you're one of the readers who've heard of prototypes in JavaScript but don't use them on a daily basis, you need not worry as we will quickly cover the basic concepts and how to apply them to JavaScript performance.

If you're one of those who know what closures, inheritance, parent and child relations, and so on are, feel you fall into the latter category and so want to skip this chapter, I would encourage you to keep reading, at least to skim through the chapter, because, we as JavaScript developers tend to forget common concepts while working with JavaScript for many years and continuously focusing on just the factors that affect our performance.

The care and feeding of function names

Take a close look at this simple function shown below and see if you spot anything unusual about this function.



```
1  /*
2  * Chapter 6 - 01
3  *
4  */
5
6  function AuthorName(author) {
7      "use strict";
8      return author;
9  }
10
```

Now, when we look at the code, what we can see is a simple function named `AuthorName` holding the `author` parameter. The function uses a `use strict` statement discussed in *Chapter 2, Increasing Code Performance with JSLint*, which forces **Developer tools** or other similar inspectors to treat any kind of warnings in that scope as errors. We then return the `author` parameter using the `return` keyword.

This looks fairly normal; however something that trips up many JavaScript developers is how the function name is structured. Notice that `AuthorName` starts with a capital A. In JavaScript, when we declare a function name with a capital letter, we are actually telling the JavaScript interpreter that we are declaring a constructor.

A constructor is simply a JavaScript function, and it works the same way as any other function. We can even print an author's name to the console using a simple `console.log` function as shown in the following screenshot using **Developer tools**:

A screenshot of a code editor window titled 'Chapter_06' and '06_02.js'. The code is as follows:

```
1  /*
2  * Chapter 6 - 02
3  *
4  */
5
6  function AuthorName(author) {
7      "use strict";
8      return author;
9  }
10
11 console.log(AuthorName('Chad Adams'));
```

If we run this in an `about:blank` **Developer tools** console or dummy HTML page with this code, we will see the console output as the name just as we would expect. The problem is that, in order to efficiently use constructors, we need to use them with the `new` keyword.

Now you may ask how we can possibly know if any of our existing JavaScript code uses constructors. Imagine a very large code base with functions everywhere; how can we check this if even the **Developer tools** option doesn't inform us that we need to use an instance using `new` rather than a `static` function call?

Luckily, there is a way. If we remember in *Chapter 2, Increasing Code Performance with JSLint*, JSLint can show us if we need to use a new keyword. I've added the preceding code sample and enabled the **console** and **browser** objects in JSLint. Check out the error presented by JSLint in the following screenshot:



As we can see from JSLint, on line 11 we're given an error, Missing 'new' as our only error, indicating that we have a constructor and we need to use it as such.

Understanding instances

Now the easy way to fix this issue is to change the name of the `AuthorName` function to camel case; that is, we change `A` to lowercase (`a`). But here we're going to indicate this as an instance, and you may ask why? Well in JavaScript, every time we write an object, a variable, a function, an array, and so on, we are creating objects.

By using instances, we keep our object usage down. In JavaScript, an instance is only counted once in memory. For example, let's say we use a `document.getElementById()` method. Every variable saved with that object is given a memory count of one but, if it's in an object we declare with the `new` keyword, that count is only counted once rather than reused for every occurrence of `getElementById()`. By using the `new` keyword, we create an instance of our constructor (in this case `AuthorName`) that allows us to reuse that function in the same manner that we typically use it.

Creating instances with 'new'

Creating a new instance is pretty easy; we can simply call a new instance to run a function, as shown in the following screenshot, using the `new` keyword in our `console.log` function on line number 11:



```
1  /*
2   * Chapter 6 - 03
3   *
4   */
5
6  function AuthorName(author) {
7      "use strict";
8      return author;
9  }
10
11 console.log(new AuthorName('Chad Adams'));
```

If we run this code inside a blank page or a simple HTML page, we will see that our log doesn't output the way we expect. But we can see an object return `AuthorName {}` in the **Console** panel of the **Development tools** in Chrome. What this shows us is that we are actually logging a new instance of an object, but not the author's name.

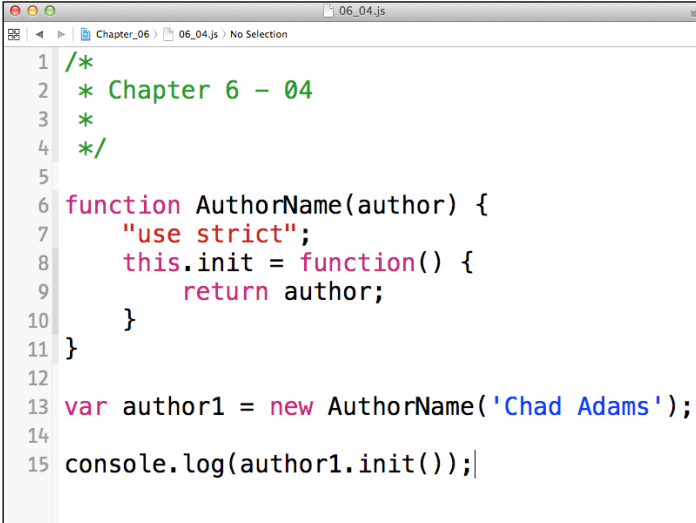
In order to properly display this name, we will need a keyword to declare a reference to the instance of this constructor. To do that, we will use the `this` keyword; in JavaScript, `this` is a reference to the exact point in execution in a scope.

The `this` keyword in JavaScript refers to the scope and variable that exists at that point of script execution when it is used. For example, when you use a `this` keyword in a function, it can reference a variable that is also in the same scope (or inside a function). By using a `this` keyword, we can point to variables and objects at a certain point in a code's execution.

A **scope** is simply a block of JavaScript code with its own variables and properties. Scopes can include a global-level scope of a single JavaScript object, meaning an entire JavaScript file, a function-level scope where variables and properties are set inside a function, or, as discussed earlier, a constructor since a constructor is a function.

Let's rewrite our `AuthorName` constructor with `this` keyword so that we can reference our scope and print our value to the **Console** panel. We will need to create an initializer inside our constructor in order to get our scoped variables returned. An initializer (sometimes called an `init` function) specifies certain variables inside our constructor and assigns properties on creation.

Here, we create a variable with a `this` keyword prefixed to indicate that we are referencing our instance inside our constructor followed by our function called `init`, which equals a function, just as we would use a variable to declare a function. Let's take a look at this in the code shown in the next screenshot:

A screenshot of a code editor window titled '06_04.js'. The editor shows the following JavaScript code:

```
1  /*
2  * Chapter 6 - 04
3  *
4  */
5
6  function AuthorName(author) {
7      "use strict";
8      this.init = function() {
9          return author;
10     }
11 }
12
13 var author1 = new AuthorName('Chad Adams');
14
15 console.log(author1.init());
```

Take a look at line numbers 13 and 15; on 13 we declare a variable `author1`, with a new `AuthorName` constructor having `Chad Adams` as the string parameter. In this example `author1` is an instance of the `AuthorName` constructor with `Chad Adams` as the only parameter.

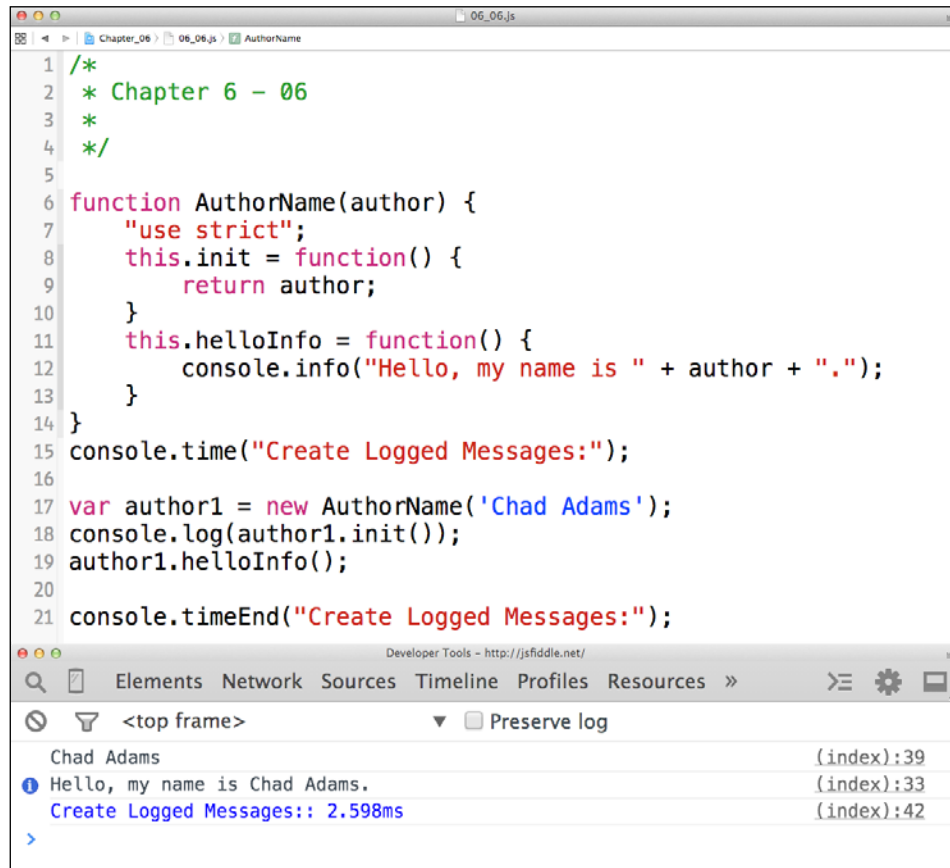
Also notice that, on line number 15 in our `console.log`, we have an `init()` function that is an inner function of our constructor. We can create other functions in our constructor as well, for instance printing a custom log message like this as shown in the next screenshot:

A screenshot of a code editor window titled '06_05.js'. The editor shows the following JavaScript code:

```
1  /*
2  * Chapter 6 - 05
3  *
4  */
5
6  function AuthorName(author) {
7      "use strict";
8      this.init = function() {
9          return author;
10     }
11     this.helloInfo = function() {
12         console.info("Hello, my name is " + author + ".");
13     }
14 }
15
16 var author1 = new AuthorName('Chad Adams');
17
18 console.log(author1.init());
19
20 author1.helloInfo();
```

As we can see on line number 11, we have now added a `helloInfo()` function, scoped to our `AuthorName` constructor, that prints out a custom message using the `author` parameter. Then, on line number 20, we call this outside a `console.log` by simply calling the function of the constructor, which has its own `console.info` function wrapped inside.

This is helpful in keeping our logic confined to a single object inside our code base and keeps our code nicely organized. This is called object orientation; it's great for reusing code but might cause issues with performance in JavaScript. Let's try an example. Here, we have two examples of the same code, each wrapped in a `console.time` and `console.timeEnd` function. The following screenshot shows our reviewed code and the resulting time to render the code:



The screenshot shows a web browser window with a code editor and developer tools. The code editor displays the following JavaScript code:

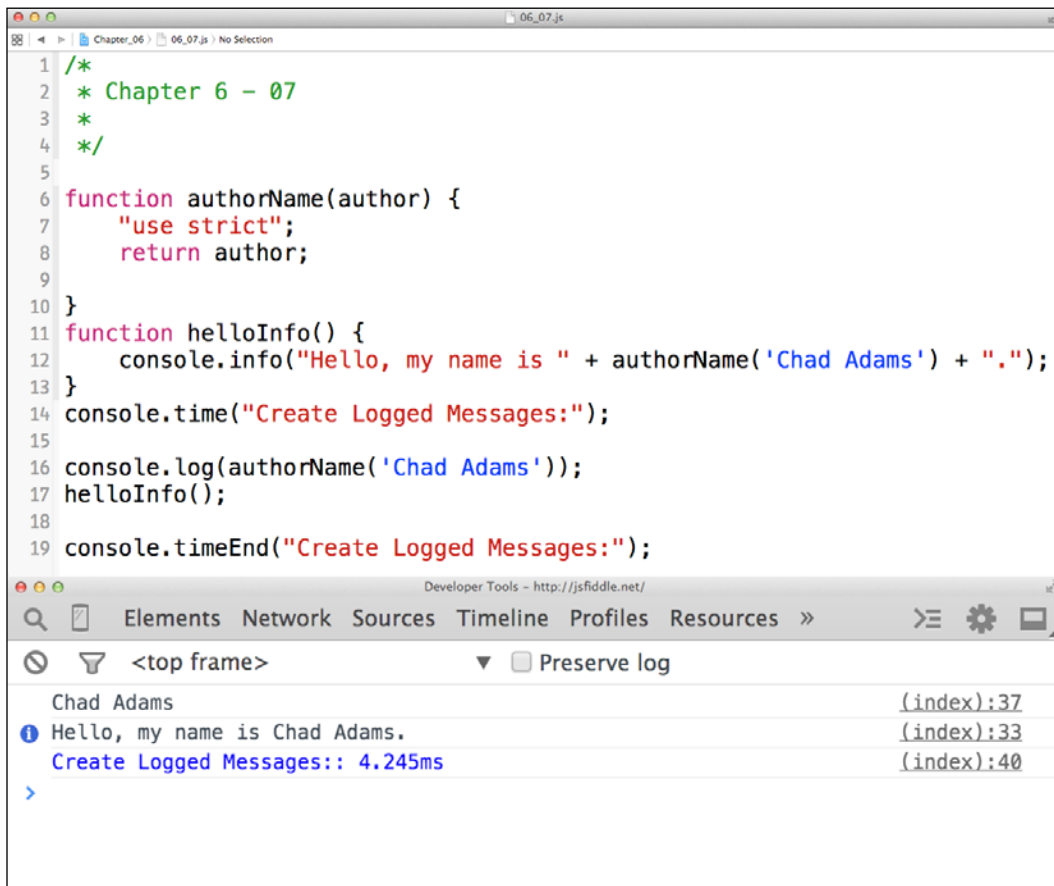
```
1 /*
2  * Chapter 6 - 06
3  *
4  */
5
6 function AuthorName(author) {
7     "use strict";
8     this.init = function() {
9         return author;
10    }
11    this.helloInfo = function() {
12        console.info("Hello, my name is " + author + ".");
13    }
14 }
15 console.time("Create Logged Messages:");
16
17 var author1 = new AuthorName('Chad Adams');
18 console.log(author1.init());
19 author1.helloInfo();
20
21 console.timeEnd("Create Logged Messages:");
```

The developer tools console shows the following output:

Message	Time
Chad Adams	(index):39
Hello, my name is Chad Adams.	(index):33
Create Logged Messages:: 2.598ms	(index):42

So our total time here is roughly 2.5 milliseconds. This isn't too bad, but now let's see what happens if we use simple non-constructor functions and what the speed of rendering the same output would be. As shown in the following screenshot, I've pulled apart our constructor and created two separate functions.

I've also called the main `authorName` function within the secondary function in the exact same manner as on our `console.log` function to print an author's name. Let's run the code shown updated in the next screenshot, and see if this runs faster or slower than our constructor methods. However, do keep in mind that, depending on our system's speed and browser, the results may vary.



The screenshot shows a web browser window with a developer console open. The console displays the following JavaScript code and its execution results:

```
1  /*
2  * Chapter 6 - 07
3  *
4  */
5
6  function authorName(author) {
7      "use strict";
8      return author;
9  }
10
11 function helloInfo() {
12     console.info("Hello, my name is " + authorName('Chad Adams') + ".");
13 }
14 console.time("Create Logged Messages:");
15
16 console.log(authorName('Chad Adams'));
17 helloInfo();
18
19 console.timeEnd("Create Logged Messages:");
```

The console output shows the following results:

- `Chad Adams` (index):37
- `Hello, my name is Chad Adams.` (index):33
- `Create Logged Messages:: 4.245ms` (index):40

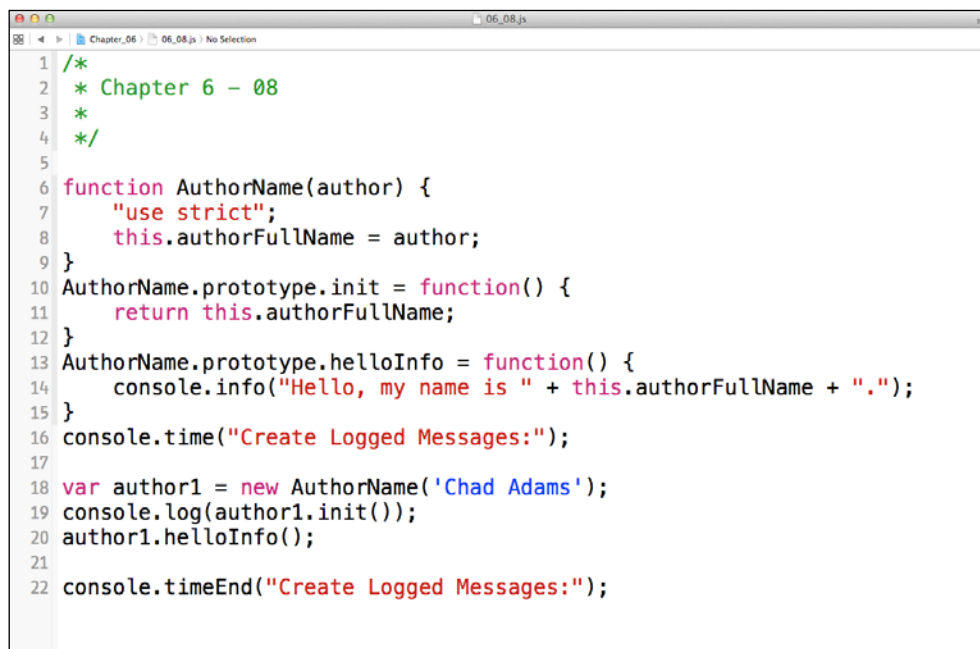
So, with static functions our results hover right around 4 milliseconds, which is longer than our instance-built object. So, that's a great use of static functions over prototype functions in JavaScript!

Alternate constructor functions using prototypes

Here, we will learn about the concept of alternate constructor functions using prototypes.

Understanding prototypes in terms of memory

We've covered how to create instance functions inside a constructor, and we've also learned about scope inside one as well using the `this` keyword. But, there is one more thing to cover: the ability to append a constructor with another instance method outside the constructor, which is helpful in many ways. First, it allows us, as developers, to create functions outside the pre-written constructor if needed. Next, it also keeps our memory usage small. Before diving into this, let's rework our constructor code to use prototypes, as shown in the next screenshot:

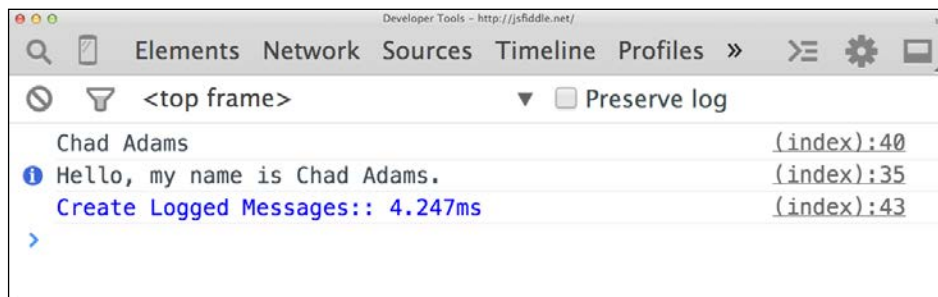


```
1  /*
2  * Chapter 6 - 08
3  *
4  */
5
6  function AuthorName(author) {
7      "use strict";
8      this.authorFullName = author;
9  }
10 AuthorName.prototype.init = function() {
11     return this.authorFullName;
12 }
13 AuthorName.prototype.helloInfo = function() {
14     console.info("Hello, my name is " + this.authorFullName + ".");
15 }
16 console.time("Create Logged Messages:");
17
18 var author1 = new AuthorName('Chad Adams');
19 console.log(author1.init());
20 author1.helloInfo();
21
22 console.timeEnd("Create Logged Messages:");
```

Now looking at this code updated, we can see that the constructor functions have been removed but pulled outside the constructor: they were then moved to prototypes of the `AuthorName` function using the same function names used earlier. Now, you can notice that, on lines 10 and 13, we can use this in our prototype function because we are referring our constructor's instance to print that instance's specific variables.

Which is faster, a prototype or a constructor function?

You may also notice that I've again added `console.time` and `console.timeEnd` functions to our function calls on lines 16 through 22. So do you think that prototypes will be faster or slower when compared to standard constructor functions? Well, here in the next screenshot we can take a look at the results:



Wow, 4.2 milliseconds to fire the prototypes when compared with 2.1 milliseconds using constructor method; what happened here? We essentially created functions after the constructor. The output is slow, but this is to be expected with prototypes, as the intention is to append a constructor with a prototype.

At this point, we may think "Oh wow I never knew that, I should never code a prototype again!" Now, before we start deleting prototypes from our project files, I want to explain scalability with prototypes. It is true that prototypes, when called for a constructor function, can be slower... *"in the small"*. What do I mean by in the small?" Well, for small uses of prototypes such as this particular example, we can see prototypes run slower than a traditional constructor method.

Now here's the rub; for larger projects, a constructor in a large-scale application may have 50 functions, 200 functions, and so on. When we call those larger constructors over and over, it gets pretty expensive in terms of memory just by calling the instance of a constructor since it has to ready all the functions contained inside.

By using prototype methods, those initial constructor calls are stored in memory once. For small uses of prototypes the performance benefits aren't visible since we use the memory in-place as if we had a simple static function but, once it's set up, it's in memory and doesn't have to be recalled or reprocessed like a static JavaScript function.

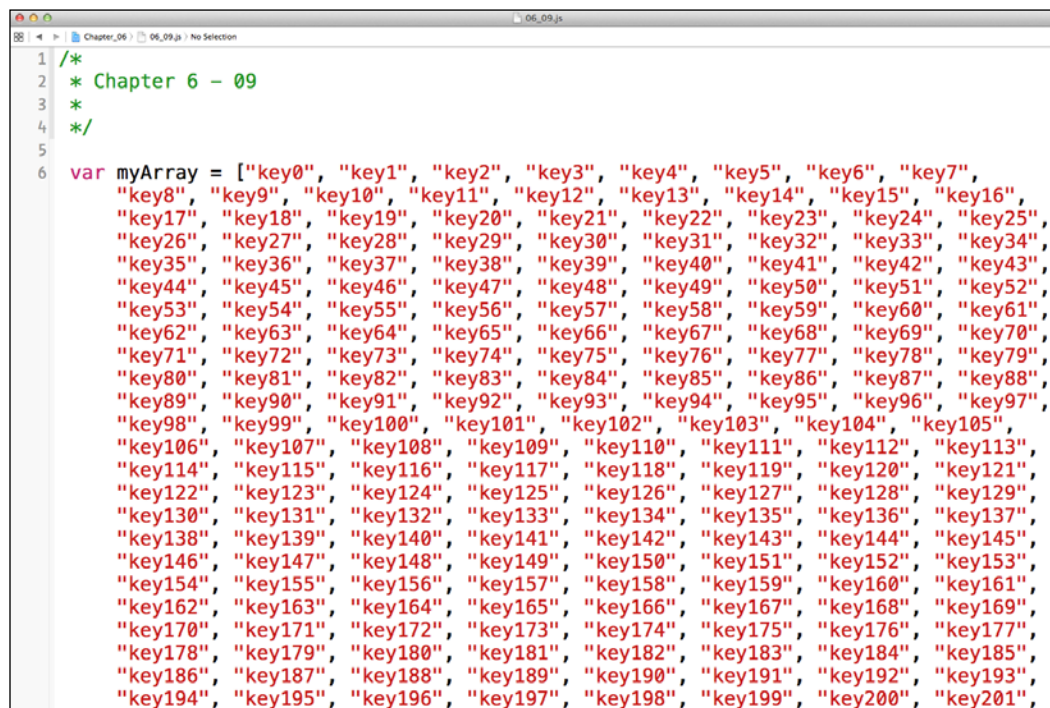
One more thing about prototypical inheritance, though performance issues can arise from its use over static functions, it can be very helpful for large codebases. If a project has scope concerns or uses libraries that may cause conflicts, consider using a namespace. This works similar to prototype classes but function like simple static functions that prepend with a namespace to prevent conflict.

Array performance

We typically don't think about arrays when dealing with performance, but it's worth mentioning a few things here. First, large arrays can be messy and performance hogs when you're trying to work with a large amount of data. Typically with arrays, we only need to worry about two things: searching and array size.

Optimizing array searches

Let's create an array that has a lot of values in it; here I've created an array called `myArray` which has 1001 values within, with a string value of key and the index of the array. You can grab the full version in the `06_09.js` file inside the `Chapter_6` folder of the example code on the Packt Publishing's website. Here's a part of the code sample of the total array:

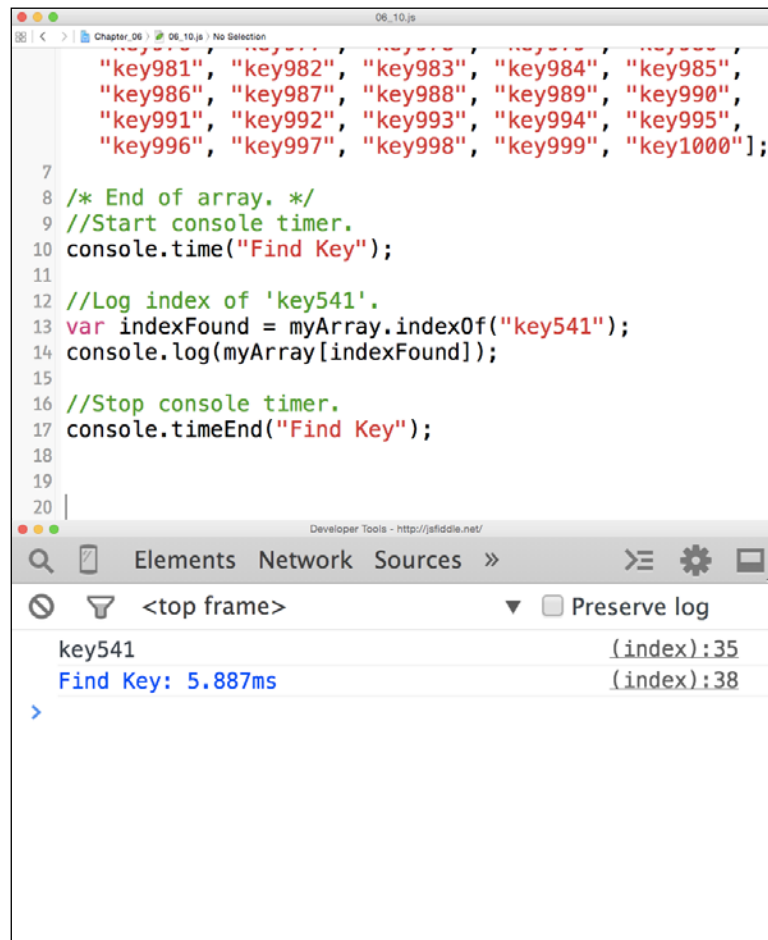


```

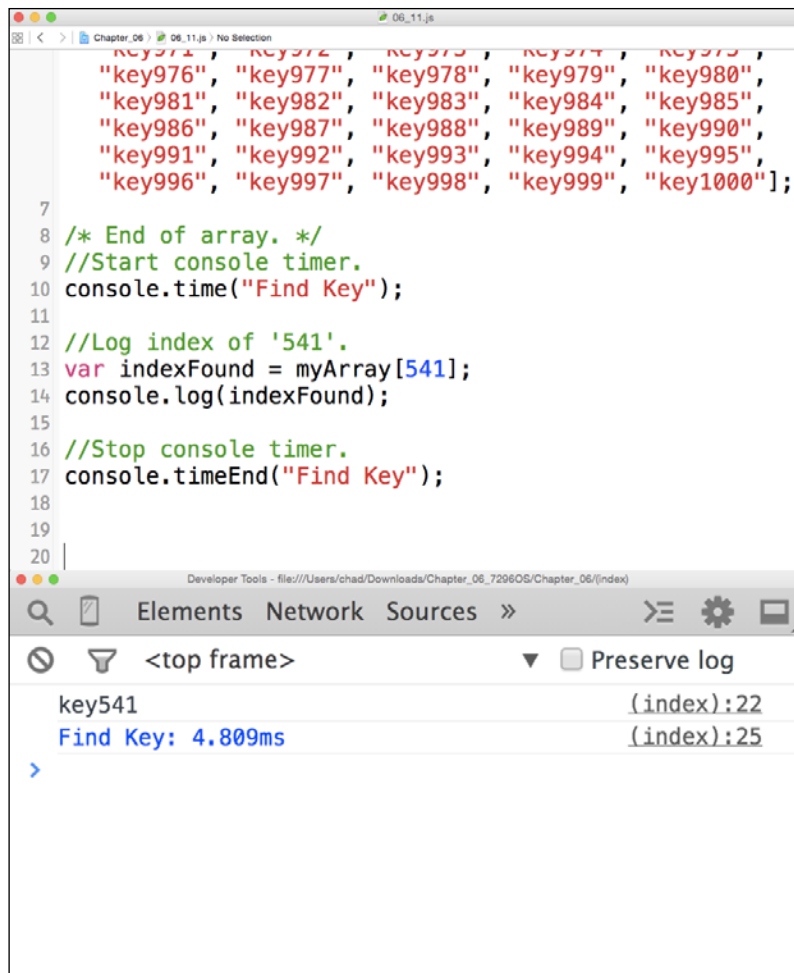
1  /*
2  * Chapter 6 - 09
3  *
4  */
5
6  var myArray = ["key0", "key1", "key2", "key3", "key4", "key5", "key6", "key7",
    "key8", "key9", "key10", "key11", "key12", "key13", "key14", "key15", "key16",
    "key17", "key18", "key19", "key20", "key21", "key22", "key23", "key24", "key25",
    "key26", "key27", "key28", "key29", "key30", "key31", "key32", "key33", "key34",
    "key35", "key36", "key37", "key38", "key39", "key40", "key41", "key42", "key43",
    "key44", "key45", "key46", "key47", "key48", "key49", "key50", "key51", "key52",
    "key53", "key54", "key55", "key56", "key57", "key58", "key59", "key60", "key61",
    "key62", "key63", "key64", "key65", "key66", "key67", "key68", "key69", "key70",
    "key71", "key72", "key73", "key74", "key75", "key76", "key77", "key78", "key79",
    "key80", "key81", "key82", "key83", "key84", "key85", "key86", "key87", "key88",
    "key89", "key90", "key91", "key92", "key93", "key94", "key95", "key96", "key97",
    "key98", "key99", "key100", "key101", "key102", "key103", "key104", "key105",
    "key106", "key107", "key108", "key109", "key110", "key111", "key112", "key113",
    "key114", "key115", "key116", "key117", "key118", "key119", "key120", "key121",
    "key122", "key123", "key124", "key125", "key126", "key127", "key128", "key129",
    "key130", "key131", "key132", "key133", "key134", "key135", "key136", "key137",
    "key138", "key139", "key140", "key141", "key142", "key143", "key144", "key145",
    "key146", "key147", "key148", "key149", "key150", "key151", "key152", "key153",
    "key154", "key155", "key156", "key157", "key158", "key159", "key160", "key161",
    "key162", "key163", "key164", "key165", "key166", "key167", "key168", "key169",
    "key170", "key171", "key172", "key173", "key174", "key175", "key176", "key177",
    "key178", "key179", "key180", "key181", "key182", "key183", "key184", "key185",
    "key186", "key187", "key188", "key189", "key190", "key191", "key192", "key193",
    "key194", "key195", "key196", "key197", "key198", "key199", "key200", "key201",
  ]
  
```

There are two ways to look up a value in an array; the first uses the `indexOf()` function, which is an array-specific function that looks up each value and returns the index of that searched value. The other way is to specify the index value directly, which returns the value (assuming we know the index of the value we need).

Let's try an experiment where we are going to use a pre-made `myArray` of 1001 values and iterate through them with the `indexOf()` function, and then again with just an array. We have appended the code after our `myArray`, and we've wrapped this code block in `console.time` and `console.timeEnd` functions, as shown here with the time rendered in Chrome **Developer tools**:



This shows that our result for searching that large array was roughly 5.9 milliseconds. Now, for our comparison, I'm going to keep our `indexFound` variable even though we can simply specify the index of the array value we want. We will also search using the same index value, which is 541. Let's update our code as shown here and view our results in Chrome **Developer tools**:



The screenshot shows a web browser window with a JavaScript file named '06_11.js' open. The code defines an array of 25 keys, from 'key976' to 'key1000'. It then uses `console.time("Find Key");` to start a timer, followed by `var indexFound = myArray[541];` and `console.log(indexFound);` to log the value at index 541. Finally, it uses `console.timeEnd("Find Key");` to stop the timer. The browser's developer tools are open, showing the 'Sources' tab with the file loaded. The 'Console' tab is active, displaying the output of the code: 'key541' and 'Find Key: 4.809ms'. The 'Elements' and 'Network' tabs are also visible.

```
key976, key977, key978, key979, key980,
key981, key982, key983, key984, key985,
key986, key987, key988, key989, key990,
key991, key992, key993, key994, key995,
key996, key997, key998, key999, key1000];
7
8 /* End of array. */
9 //Start console timer.
10 console.time("Find Key");
11
12 //Log index of '541'.
13 var indexFound = myArray[541];
14 console.log(indexFound);
15
16 //Stop console timer.
17 console.timeEnd("Find Key");
18
19
20
```

Developer Tools - file:///Users/chad/Downloads/Chapter_06_7296OS/Chapter_06/(index)

Elements Network Sources »

<top frame> Preserve log

key541 (index):22

Find Key: 4.809ms (index):25

It looks like our results trim our index searching performance time by quite a bit. So, when you're structuring arrays in JavaScript, only use `indexOf` if you need to and try to directly call the index if possible. So why was the time output so different? It's simple; in this second example, we indicated the position of the array manually rather than having JavaScript look up the key on its own. This sped up the JavaScript interpreter as it iterated through our array and provided a value.

Summary

In this chapter, we learned about the proper use of constructors. We learned about instances in JavaScript using the `new` keyword, and found that we can speed up static code with constructors while scoping our code at the same time.

We learned about prototypes and how they scale well for large applications while adding little value for smaller projects. Finally, we also learned about searching arrays and about performance loss with arrays by using the `indexOf` function.

In the next chapter, we are going to look at how to write our JavaScript to optimize our Document Object Model for our projects.

7

Hands off the DOM

In this chapter, we will review the DOM in relation to writing high-performance JavaScript, and see how to optimize our JavaScript to render our web applications visibly faster.

We will also take a look at JavaScript animations and test their performance against modern CSS3 animations; we will also test for paint redraw events in the DOM and quickly test for scrolling events attached to a page that may affect performance.

We will cover the following topics in the chapter:

- Why worry about the DOM?
- Don't we need a MV-whatever library?
- Creating new objects using the `createElement` function
- Animating elements
- Understanding paint events
- Pesky mouse scrolling events

Why worry about the DOM?

The **Document Object Model (DOM)** is how our HTML content is presented in our web browser. It's not quite the same as the source code; the DOM is the live updated version of our source code as we make updates to a web application's page in a web browser.

We can say that fast, optimized JavaScript will certainly help our applications run and perform better, as we learned in previous chapters. But it's important to understand that the DOM is just as important to JavaScript performance as understanding how to optimize a `for` loop.

In the early days of the Web, we as web developers didn't think about the DOM too much. If we think about how far JavaScript has come, we can see that many changes have come to the world of web development. If we reminisce about the pre-Google days of the web, we know that websites were pretty simplistic, and user interaction was mainly limited to hyperlink tags and an occasional JavaScript `window.alert()` function to show some form of application interaction.

As time passed, we encountered Web 2.0, or rather the point where **Asynchronous JavaScript and XML (AJAX)** came into being. If you're not familiar with AJAX, I would like to sum it up: AJAX web applications allow developers to pull content from external sources, typically XML files, (this is the X in AJAX).

With AJAX, website content suddenly became dynamic, meaning developers didn't have to rely on backend technologies to refresh a web page with updated data. Suddenly, a need for stronger JavaScript came into play. Businesses and their clients no longer wanted to have a website responding with page flashes (or sites that used backend technologies to update the page with a `POST` submission method), all the more so with sites such as Google Maps and Gmail seemingly pushing the idea of the web as a platform for software rather than a desktop operating system.

Don't we need an MV-whatever library?

Today, we have frameworks that help with the heavy lifting of some of this type of application; AngularJS, Backbone.js, Knockout.js, and jQuery are a few libraries that come to mind.

For this book, however, we will stick to vanilla JavaScript for two reasons. The first reason is that entire books are dedicated to many of these libraries and talk about performance and various levels of experience, all of which are good but beyond the scope of this book. The second reason is that most developers typically don't need these libraries to build a project.

Remember that all the JavaScript libraries mentioned here, as well as those found on the Web, are again all JavaScript! For most projects, we shouldn't need a library to make a project the way we want to build it; moreover, many of these libraries come with extra code.

What I mean by this is that the libraries come with features that might not be needed in a given project and, unless a library is modular, it's difficult to use it without removing features that aren't needed. This is even harder if you're working in a team environment where others may be using a shared library for certain areas of the application that may use some features, but not all of them.

We'll look into mobile JavaScript performance later in *Chapter 9, Optimizing JavaScript for iOS Hybrid Apps*. We will find these libraries become even more of a burden. Now with that said, let's look at some common ways to break the DOM, and what we can do to make it perform better.

Creating new objects using the `createElement` function

Here, we will learn to create new objects using the `createElement` function along with the following three topics:

- Working around the `createElement` function
- Working with the `createElement` function
- When to use the `createElement` function

Working around the `createElement` function

In JavaScript, we can create new page elements using the `document.createElement()` function and text objects to place inside our generated elements using the `document.createTextNode()` function. Typically, creating new elements to inject into our DOM can be a bit of a drain on rendering resources as well as interaction performance if done with multiple generated elements.

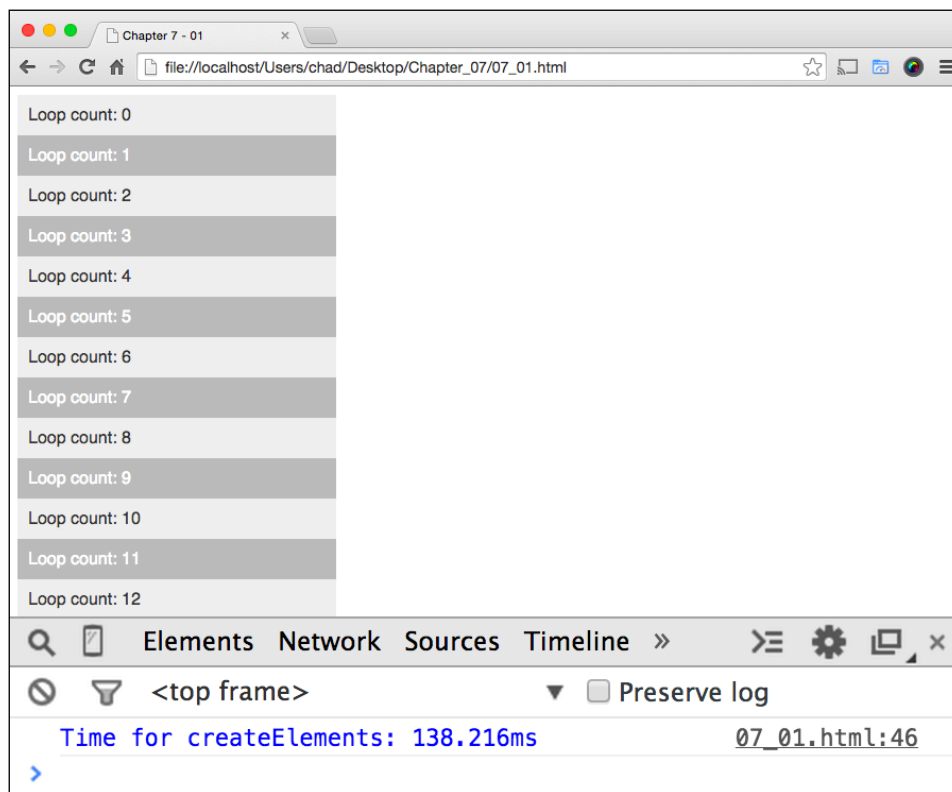
Working with the createElement function

Let's test how well the `createElement` function renders content to a screen. Here's our test: we are going to create a table with a lot of data using a `for` loop. We will populate a table cell with a text object with the count of the iteration of our `for` loop. Then, we will look at an alternate version creating the same effect with a different code implementation, and compare both. Let's take a look at the first option using the `createElement` function shown as follows:

A screenshot of a code editor window titled "07_01.html". The editor shows a mix of HTML and JavaScript code. The HTML part includes a DOCTYPE declaration, a head section with a meta tag for content type and charset, a title "Chapter 7 - 01", and a style block. The style block uses `#datainsert tr:nth-of-type(odd) td` and `#datainsert tr:nth-of-type(even) td` selectors to style table rows. The body contains a comment "

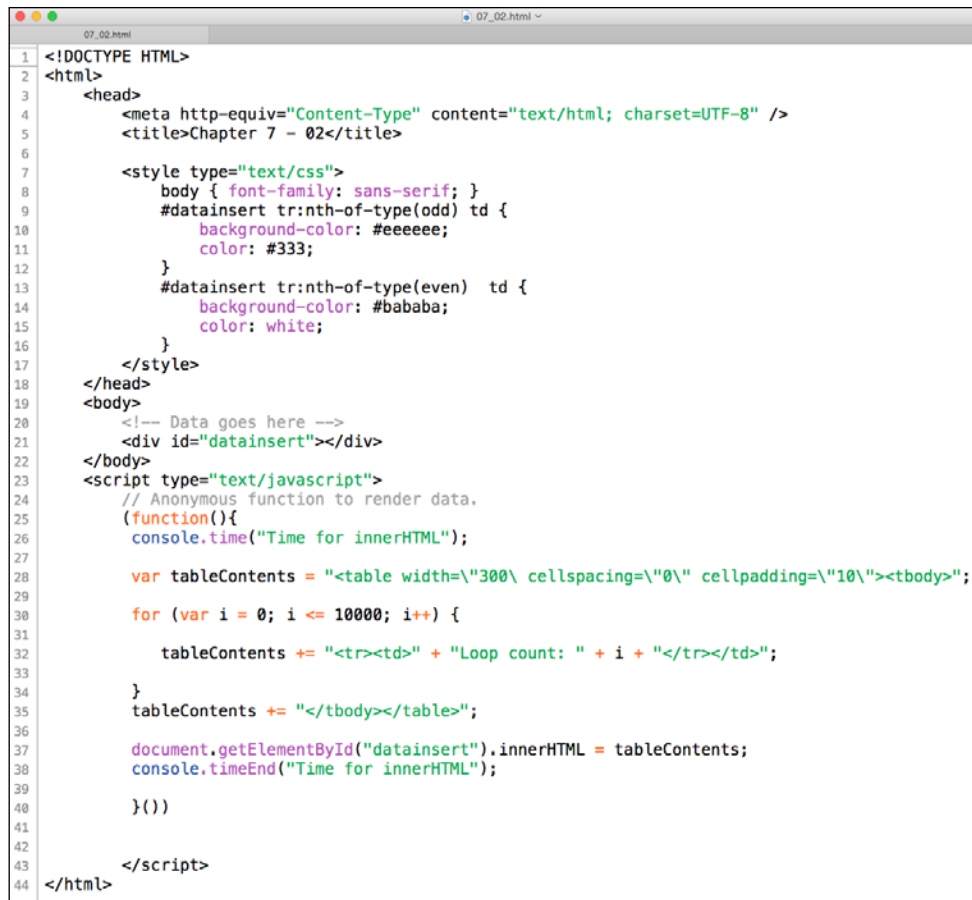
Here, we have a simple HTML5 page with some formatting CSS styles in the head section and an empty placeholder `div` element on line number 21 with an `id` set as `datainsert`. On line 25, we have an anonymous function to run as soon as it's loaded into the browser; also on line 26, we start a `console.time` function to start counting how long our JavaScript performs. We then create a table element variable called `tableElem` on line 27; on lines 28 through 31, we set some attributes to help style the formatting of our table.

Then on line 33, we start our `for` loop; in the scope of our `for` loop we create a table row element, a table cell element, and a text node to insert text into our generated table cell, starting with the `cellContent` variable on line 35, the `tableTr` variable on line 36, and the `tableTd` variable on line 37. On lines 39-41, we append the table with our generated cell and continue the loop for 10000 times. Lastly, we append the table element to our `datainsert` `div` element on the page to render our content. Let's run this in our browser and see how long it takes for the content to render using Chrome Developer tools options.



As we can see, this took quite a bit of processing time, roughly 140 milliseconds in Chrome, which is a pretty lengthy render. You can consider doing something like this in building a messaging client or displaying data from JSON. Whatever the case, the cost of using the `createElement` function is quite large and should only be used in small portions.

The other way to generate data on a table like this, but without the use of the `createElement` function, is to use the `innerHTML` property. This property provides a simple way to completely replace the contents of an element and assign values in the same manner as assigning value to a variable. When the `innerHTML` property is used, you can change the page's content without refreshing the page. This can make your website feel quicker and more responsive to user input. This property can also be appended using the `+=` append operator. Knowing this, we can structure our code base in a slightly different way. What we are doing is shown in the following screenshot:

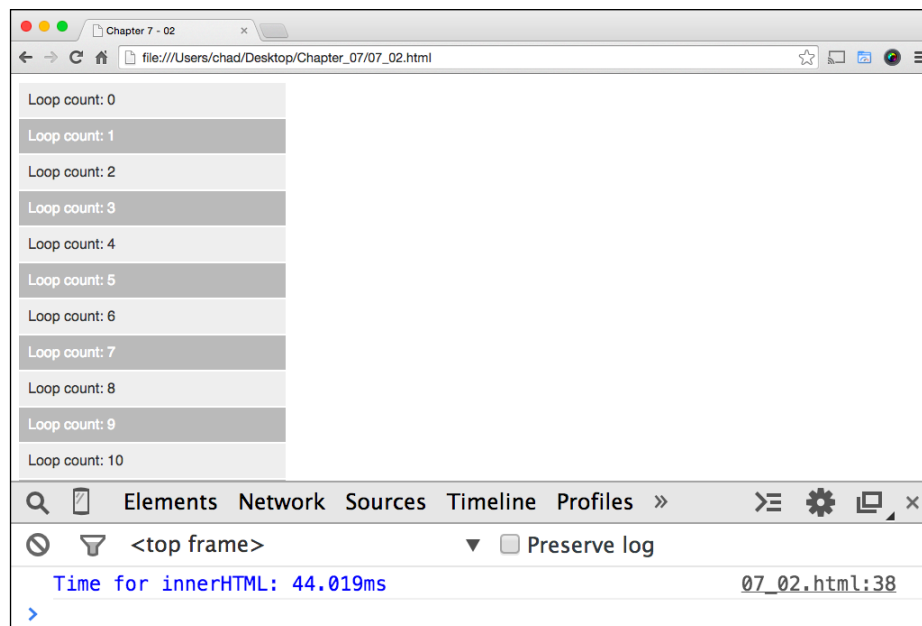


```
1 <!DOCTYPE HTML>
2 <html>
3   <head>
4     <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
5     <title>Chapter 7 - 02</title>
6
7     <style type="text/css">
8       body { font-family: sans-serif; }
9       #datainsert tr:nth-of-type(odd) td {
10         background-color: #eeeeee;
11         color: #333;
12       }
13       #datainsert tr:nth-of-type(even) td {
14         background-color: #bababa;
15         color: white;
16       }
17     </style>
18   </head>
19   <body>
20     <!-- Data goes here -->
21     <div id="datainsert"></div>
22   </body>
23   <script type="text/javascript">
24     // Anonymous function to render data.
25     (function(){
26       console.time("Time for innerHTML");
27
28       var tableContents = "<table width='300' cellspacing='0' cellpadding='10'><tbody>";
29
30       for (var i = 0; i <= 10000; i++) {
31
32         tableContents += "<tr><td>" + "Loop count: " + i + "</tr></td>";
33
34       }
35       tableContents += "</tbody></table>";
36
37       document.getElementById("datainsert").innerHTML = tableContents;
38       console.timeEnd("Time for innerHTML");
39
40     })()
41
42   </script>
43 </html>
```

The layout for this should be pretty similar to our `createElement` function example. On line 21, we have the same `datainsert` div; on line 25 our `Anonymous` function is started off. Now on line 28, we see something quite different; here, we can see the start of a string variable called `tableContents`, with the start of an HTML table with the same properties as that of the preceding example set to it. This is just like what we did using the `createElement` function except that we used just a JavaScript string of HTML markup rather than a DOM object this time.

Next on line 30, we start our `for` loop and append the `tableContents` string with an appended string adding in our table row and table cell, with the `for` loop's iteration count inserted into the cell, again counting 10,000 times.

When the loop is finished on line 35, we append our string with the closing brackets for our table. Finally on line 37 and 38, we use the `innerHTML` property and write our table into the `innerHTML` property of the `datainsert` div element. Let's run this example in a browser and take a look at its processing time.



This time our table's render time is roughly 40 milliseconds, which is almost four times faster than we would get if we used the `createElement` function. Now that's a great speed improvement! And it's even visually faster in Chrome as well.

When to use the `createElement` function?

Though the `createElement` function is slow, on occasion it can be more helpful in generating HTML through a complex layout, where a complex application generates many more elements than an `innerHTML` property can be styled to accommodate.

If this is the case, this is done more for convenience and usability for the development team when modifying the element's type rather than for updating a full string to fit the needs of the application. In any case, if you need to create HTML elements, the `innerHTML` property is always faster.

Animating elements

One of the more impressive uses of JavaScript came around the *Web 2.0* age of JavaScript while AJAX was gaining popularity; another interesting idea came about in the form of JavaScript animations. These are animations that are created by simply iterating over and over an element's styles that are left- and top-positioned using a `setInterval` function, and then dismissing it after the element reached its end point. This allows the div to appear to tween or animate on the page itself.

Animating the old-fashioned way

Most JavaScript developers are familiar with doing animations using jQuery, the popular DOM manipulation library for JavaScript, using the `animate` function to create DOM animations. But, as we are talking about pure JavaScript in this book, let's take a look at an example of how to build this from the ground up. Check out the code in the following screenshot:

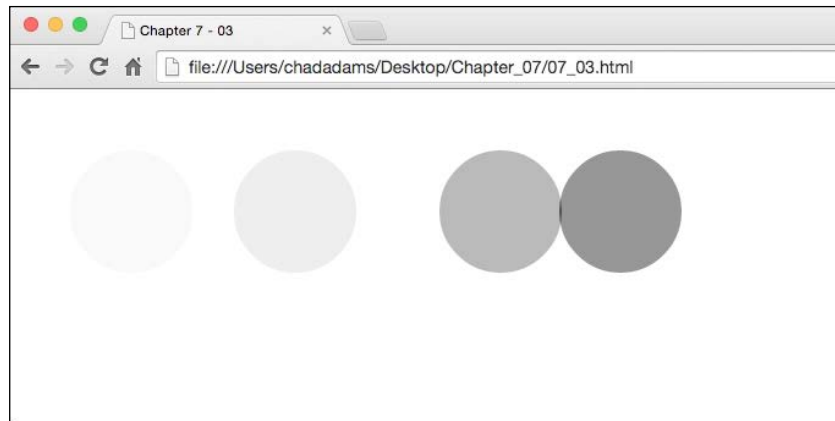
```

1 <!DOCTYPE HTML>
2 <html>
3   <head>
4     <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
5     <title>Chapter 7 - 03</title>
6
7     <style type="text/css">
8       body { font-family: sans-serif; }
9       #dot {
10         width: 100px;
11         height: 100px;
12         -webkit-border-radius: 50px;
13         background-color: #333;
14         position: absolute;
15         left: 50px;
16         top: 50px;
17       }
18     </style>
19   </head>
20   <body>
21     <!-- Data goes here -->
22     <div id="dot"></div>
23   </body>
24   <script type="text/javascript">
25     // Anonymous function to render data.
26     (function(){
27       var dot = document.getElementById("dot");
28       var i = 50;
29
30       console.time("JavaScript Animation");
31       var interval = setInterval(function(){
32         i++;
33         dot.style.left = i + "px";
34         if(i === 450) {
35           clearInterval(interval);
36           console.timeEnd("JavaScript Animation");
37         }
38       }, 1);
39
40     })()
41   </script>
42 </html>

```

In this example, I've simply created a WebKit-friendly animation using just JavaScript, (meaning this will only display properly in the Google Chrome and Apple Safari browsers). On line 7, we set up some basic styles including a black dot div element with the `id` set as `dot`.

Now on line 27 and 28, we declare the `dot` and `i` variables respectively. Then, on line 31, we create a variable called `interval`, which is actually a parameter passed to the `setInterval` function. In the case of this code, it's for every millisecond, which is shown on line 38. Inside the `setInterval` function we increment the `i` variable's count by 1, and update the position of the `dot` element. Finally, when the value of the `i` variable is strictly equal to 450, we dismiss our interval variable using the `clearInterval` function, which stops the `setInterval` function from processing any further. If we look at this, we can see a simple animation tween using pure JavaScript in our browser. This is shown in the following screenshot:



Now, you may think that creating a `setInterval` function in this manner may be a cause of concern, and you might be correct. Fortunately, we as developers now have an alternative when it comes to creating animations like this for our HTML5 applications!

Animating using CSS3

Let's rebuild this example using CSS3 and JavaScript only to trigger the animation. Again, we will simply style for WebKit-focused browsers, just for simplicity. Let's take a look at the updated code sample shown in the following screenshot:

```

1 <!DOCTYPE HTML>
2 <html>
3   <head>
4     <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
5     <title>Chapter 7 - 04</title>
6
7     <style type="text/css">
8       body { font-family: sans-serif; }
9       #dot {
10         width: 100px;
11         height: 100px;
12         -webkit-border-radius: 50px;
13         background-color: #333;
14         position: absolute;
15         left: 50px;
16         top: 50px;
17       }
18       /* CSS3 ANIMATION */
19       #dot.active {
20         -webkit-animation: moveDot 1000ms 0 ease both;
21         animation: moveDot 1000ms 0 ease both;
22       }
23       @-webkit-keyframes moveDot {
24         from { transform: translate3d(0, 0, 0); }
25         to { transform: translate3d(450px, 0, 0); }
26       }
27       @keyframes moveDot {
28         from { transform: translate3d(0, 0, 0); }
29         to { transform: translate3d(450px, 0, 0); }
30       }
31     </style>
32   </head>
33   <body>
34     <!-- Data goes here -->
35     <div id="dot"></div>
36   </body>
37   <script type="text/javascript">
38     // Anonymous function to render data.
39     (function(){
40       var dot = document.getElementById("dot");
41
42       console.time("JavaScript Animation");
43       setTimeout(dot.setAttribute("class", "active"), 1000);
44       console.timeEnd("JavaScript Animation");
45     })()
46   </script>
47 </html>

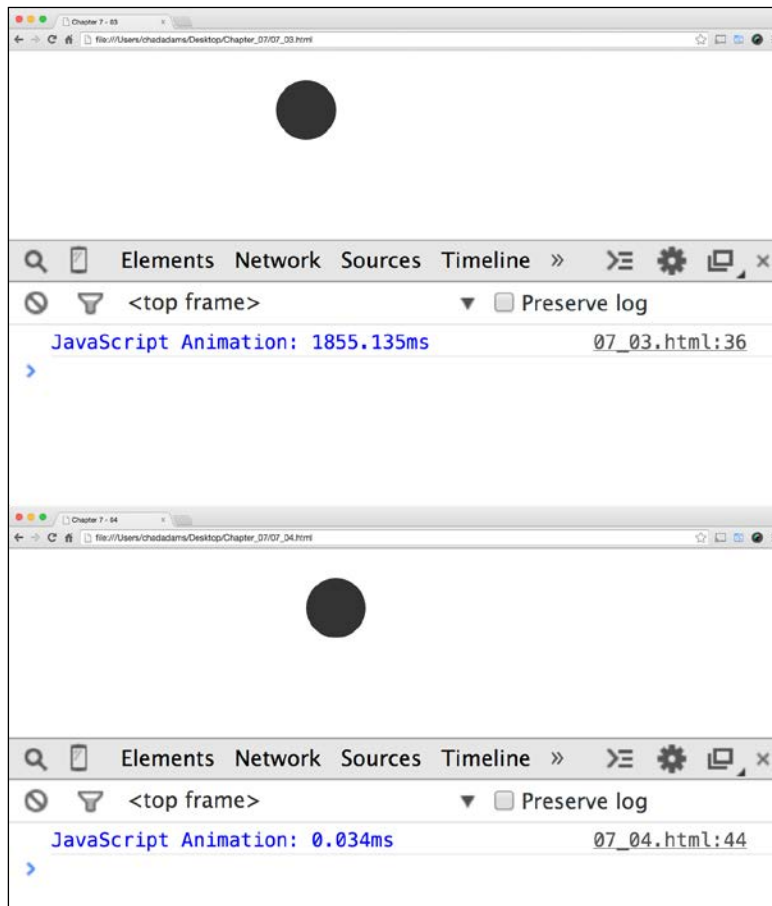
```

With this example, we can see that our JavaScript has much fewer lines, and that's a good thing; it keeps our content styles purely CSS-based rather than styling content using JavaScript logic.

Now, on the JavaScript side, we can see that we are using the same kind of Anonymous function on line 39, except that we are setting a timeout to trigger the dot element to add an active class property that triggers the animation in CSS3. This is shown on lines 19 through 30 in our example

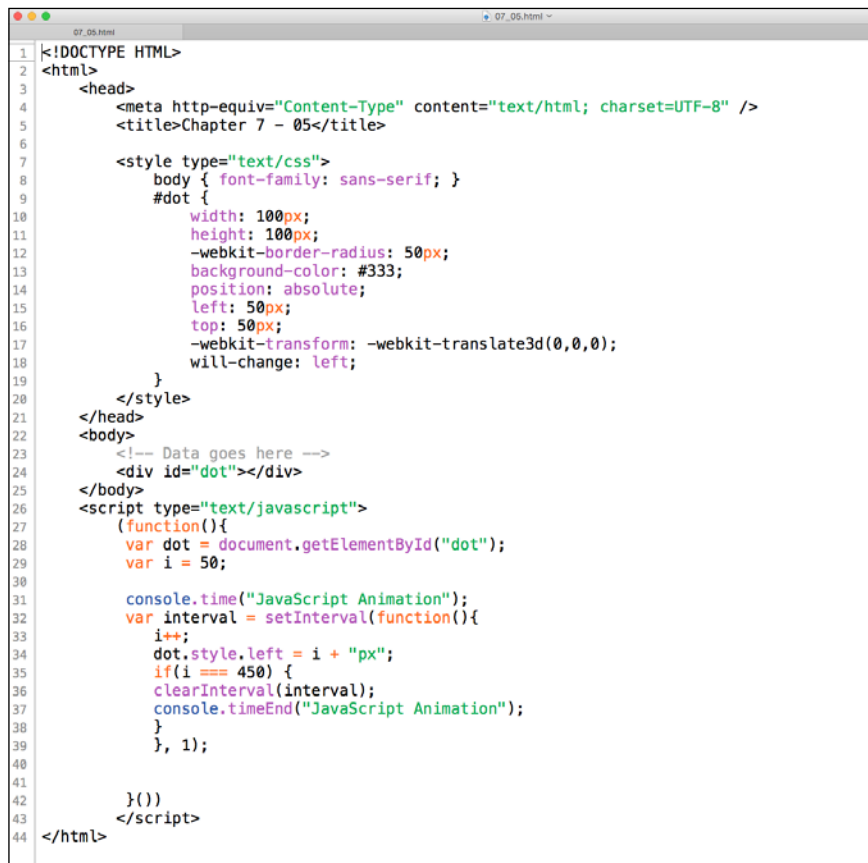
An unfair performance advantage

In many code examples in this book, I've used `console.time`, and `console.timeEnd` to review performance, and this example is no exception. You may have noticed that I've wrapped each animation example in a `time` and `timeEnd` function to measure the processing time. As we can see in the following screenshot, it's a bit one-sided:



As we can see in the preceding screenshot, the JavaScript processing time is roughly 1,900 milliseconds, and the CSS3 animation is around 0.03 milliseconds. Now, before we conclude that the CSS3 method is better, we must bear in mind that we are using CSS3 only to render the page, and JavaScript is handling only the trigger of the animation. It's still more efficient, but it should be noted that JavaScript handles less code.

Now for newer browsers, this is the recommended way of building content animations given the performance improvements seen thus far, whether made by JavaScript or not. However, some projects require older browser support, where projects may not have access to CSS3 transitions and animations, or we're upgrading a part of an application's animation while still maintaining compatibility. Here's one way of doing just that while using the same JavaScript-based animation as earlier:



```

1  <!DOCTYPE HTML>
2  <html>
3    <head>
4      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
5      <title>Chapter 7 - 05</title>
6
7      <style type="text/css">
8        body { font-family: sans-serif; }
9        #dot {
10          width: 100px;
11          height: 100px;
12          -webkit-border-radius: 50px;
13          background-color: #333;
14          position: absolute;
15          left: 50px;
16          top: 50px;
17          -webkit-transform: -webkit-translate3d(0,0,0);
18          will-change: left;
19        }
20      </style>
21    </head>
22    <body>
23      <!-- Data goes here -->
24      <div id="dot"></div>
25    </body>
26    <script type="text/javascript">
27      (function(){
28        var dot = document.getElementById("dot");
29        var i = 50;
30
31        console.time("JavaScript Animation");
32        var interval = setInterval(function(){
33          i++;
34          dot.style.left = i + "px";
35          if(i === 450) {
36            clearInterval(interval);
37            console.timeEnd("JavaScript Animation");
38          }
39        }, 1);
40
41      })()
42    </script>
43  </html>

```


Here we've modified our initial JavaScript example by updating the position of the dot element; however, we've added two CSS lines on lines 17 and 18. The first one is a `-webkit-transform` and `translate3d` property that only sets the element to not change position; in older browsers or non webkit-focused browsers, this property will be ignored. But here it's simply setting the position of the element to its initial position, which sounds silly, and in a way it is!

What this really does is tell the DOM runtime that this needs to run as a unique graphics process; it also tells the **graphics processing unit (GPU)** on the browser's device to draw this element fast! The same can be said for `will-change`, which is a similar property that does the same thing as the `translate3d` property with the exception that it's not updating the position but simply telling the GPU to redraw this element at a very high rate and to expect it to change in the DOM. Now, this practice is called adding elements to a composite layer. We will get more into composite layers in *Chapter 9, Optimizing JavaScript for iOS Hybrid Apps*. But for now, this is what we are doing here; this way, newer browsers can still get some visual speed improvements using legacy JavaScript animations.

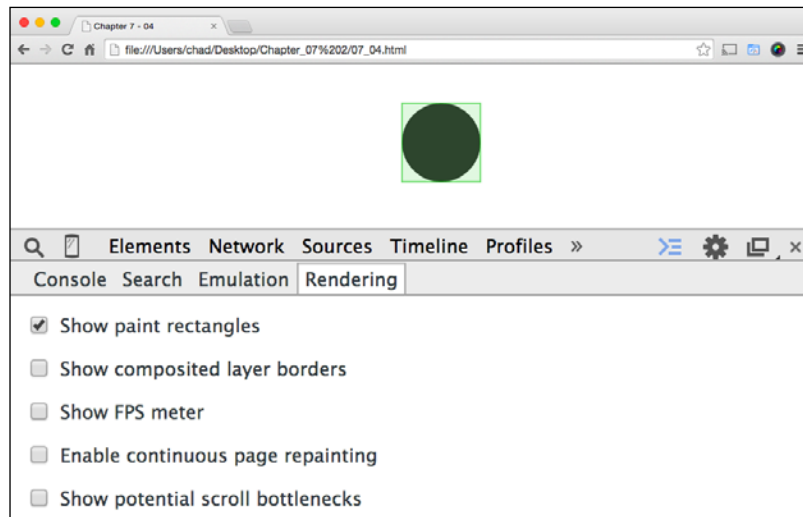
Understanding paint events

Paint events are DOM events that cause a web browser to paint the web page as the DOM is updated with JavaScript. For browsers with low memory, this can be a bit of an issue as paint events take a sizable amount of processing and graphics rendering to show updates in large quantities.

How to check for paint events?

Typically, paint events can be found in your Web Inspector's timeline view. Since paint events are displayed chronologically during a page's execution in a web browser, these appear slightly differently in Chrome's **Developer tools** options.

Open Chrome's **Developer tools** options and click the drawer icon (it's the icon next to the gear icon on the upper right of the **Developer tools** options). Next, open the **Rendering** tab in the drawer, and click **Show paint rectangles** option. Once that's finished, refresh the page. We will see the page highlighted green in various areas as the page loads. These are paint events in action as they are loaded on screen. Here's an example using our animation and showing paint rectangles enabled in Chrome's **Developer tools** options:

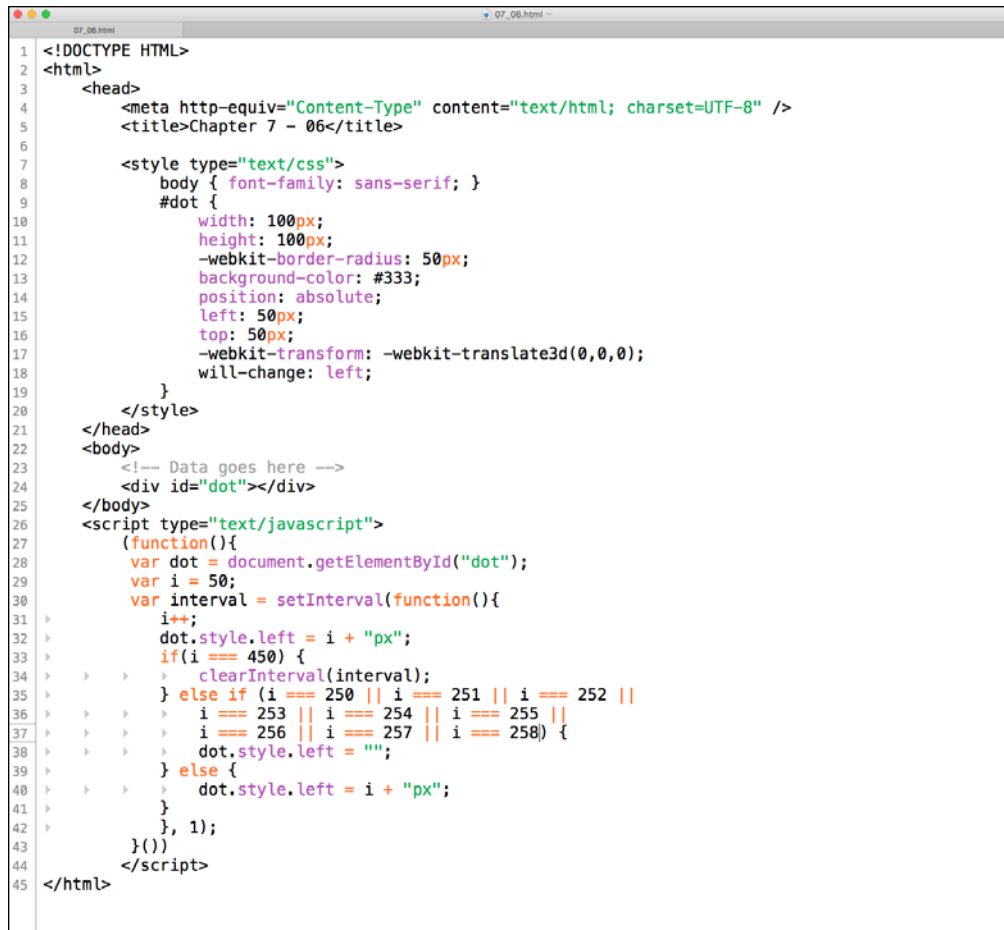


Notice how the green square appears on page load and again when the animation finishes. This happens because the DOM only repaints the browser window on page load or when an animation ends.

Occasionally, projects can create pretty complex animations using JavaScript alone. To spot errors with our JavaScript logic and ensure that a paint event isn't causing an issue, we can use the continuous page repainting feature inside Chrome's **Developer tools**.

Testing paint events

To test this, we've set up a JavaScript animation with a built-in bug as shown here:

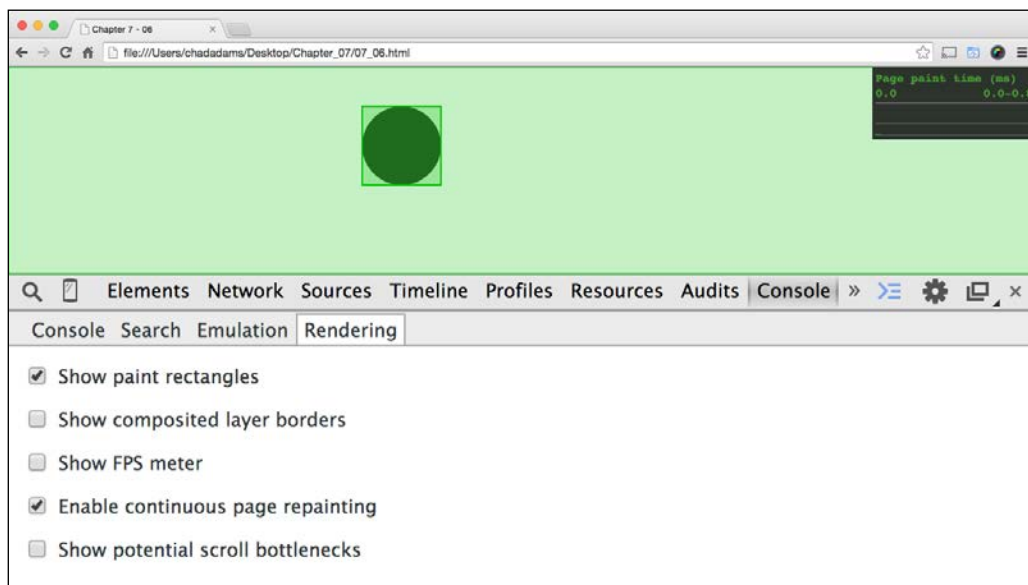
A screenshot of a code editor window titled "07_06.html". The editor displays HTML and JavaScript code. The HTML part includes a DOCTYPE declaration, a head section with a meta tag and a title "Chapter 7 - 06", and a style block for a dot element. The style block sets width, height, border-radius, background-color, position, left, top, transform, and will-change. The JavaScript part is a script block that defines a function to animate the dot's left position. It uses a setInterval to increment the left position by 1px every 50ms. A conditional else if statement checks if the increment variable i is within the range 250 to 258. If it is, the left style is removed from the dot element. The code is as follows:

```
1 <!DOCTYPE HTML>
2 <html>
3   <head>
4     <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
5     <title>Chapter 7 - 06</title>
6
7     <style type="text/css">
8       body { font-family: sans-serif; }
9       #dot {
10         width: 100px;
11         height: 100px;
12         -webkit-border-radius: 50px;
13         background-color: #333;
14         position: absolute;
15         left: 50px;
16         top: 50px;
17         -webkit-transform: -webkit-translate3d(0,0,0);
18         will-change: left;
19       }
20     </style>
21   </head>
22   <body>
23     <!-- Data goes here -->
24     <div id="dot"></div>
25   </body>
26   <script type="text/javascript">
27     (function(){
28       var dot = document.getElementById("dot");
29       var i = 50;
30       var interval = setInterval(function(){
31         i++;
32         dot.style.left = i + "px";
33         if(i === 450) {
34           clearInterval(interval);
35         } else if (i === 250 || i === 251 || i === 252 ||
36           i === 253 || i === 254 || i === 255 ||
37           i === 256 || i === 257 || i === 258) {
38           dot.style.left = "";
39         } else {
40           dot.style.left = i + "px";
41         }
42       }, 1);
43     })()
44   </script>
45 </html>
```

Much of this should seem pretty similar to earlier animations we've built in this chapter. But if we take a look at lines 35 through 38, we can see that we have a conditional `else if` statement checking to see if our increment variable `i` is within the 250-258 range count; if so, the `left` style is removed from the `dot` element.

When we run this, we should encounter a flicker right the animation hits this point. We can verify whether this is truly a JavaScript issue by enabling continuous page repainting in Chrome's **Developer tools**.

To do this, open the **Developer tools** options, open the drawer, and click the **Rendering** tab in the drawer. Then we can check the **Enable continuous page repainting** and **Show paint rectangles** options. When we do this, our web page should show a green overlay and display an information box in the upper right of our browser window. This is shown in the following screenshot:



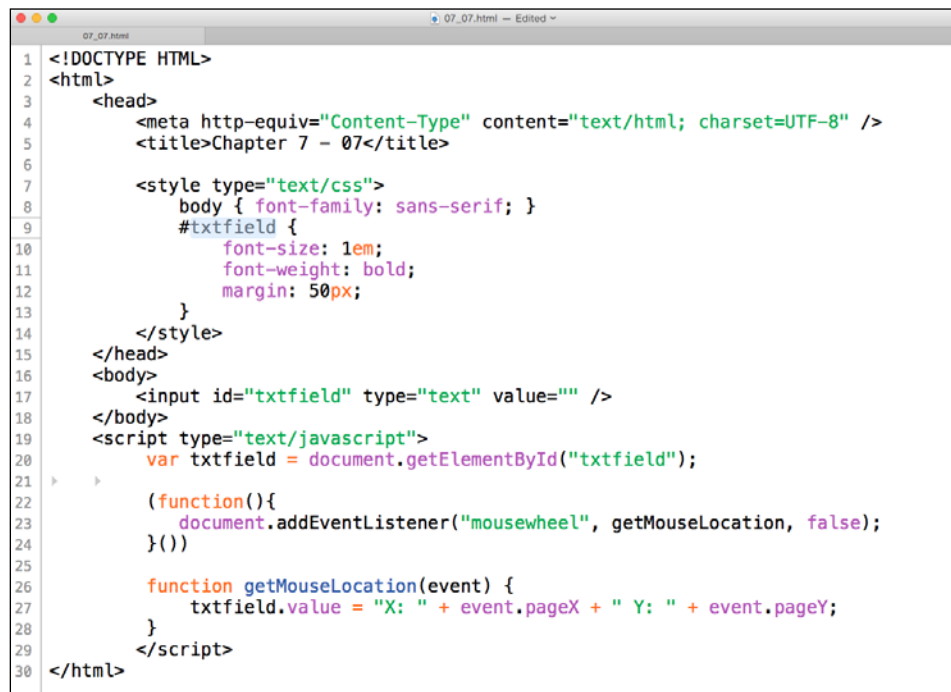
Now, when we reload the page and the animation is replayed, our dot element should show a green box drawn around it during the whole animation this time. This is Chrome forcing the page to constantly redraw as the animation updates. As we can see, the box is still on the dot even when we hit our premade bug, indicating a JavaScript issue. If this was a true paint issue, the box would disappear when a redraw issue occurs.

Pesky mouse scrolling events

Paint events (or a lack thereof) are not the only issues when it comes to web application performance when you're working with JavaScript. Scrolling events applied to a browser window or document can cause havoc on an application; it's never a good idea to continuously trigger events by scrolling a mouse, let alone multiple events.

If we're coding an application, we know whether our application has one or many events added. But if we are handed a web application to update, there is a tool in Chrome's **Developer tools** that lets us visually check for scroll events.

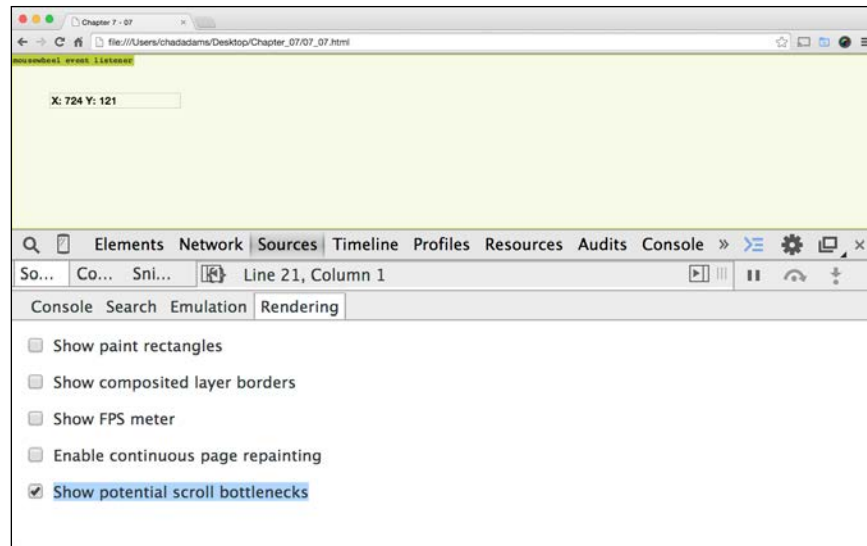
Let's create a quick example to show how this feature works and what it looks for when trying to optimize the DOM interface. For this, I've created a `mousewheel` event that will capture the X and Y coordinates of a mouse pointer's position with respect to the page, and print that to an input field with an `id` set as `txtfield`; it will trigger every time I move the mouse wheel. Let's take a look at the following code sample:

A screenshot of a code editor window titled "07_07.html - Edited". The editor shows a mix of HTML and JavaScript code. Line numbers 1 through 30 are visible on the left. The code includes a DOCTYPE declaration, an HTML structure with a head and body, a CSS style block for an input field with id "txtfield", and a JavaScript script block. The JavaScript code defines a `getMouseLocation` function and adds a `mousewheel` event listener to the document that calls this function. The function updates the value of the `txtfield` input with the current pageX and pageY coordinates.

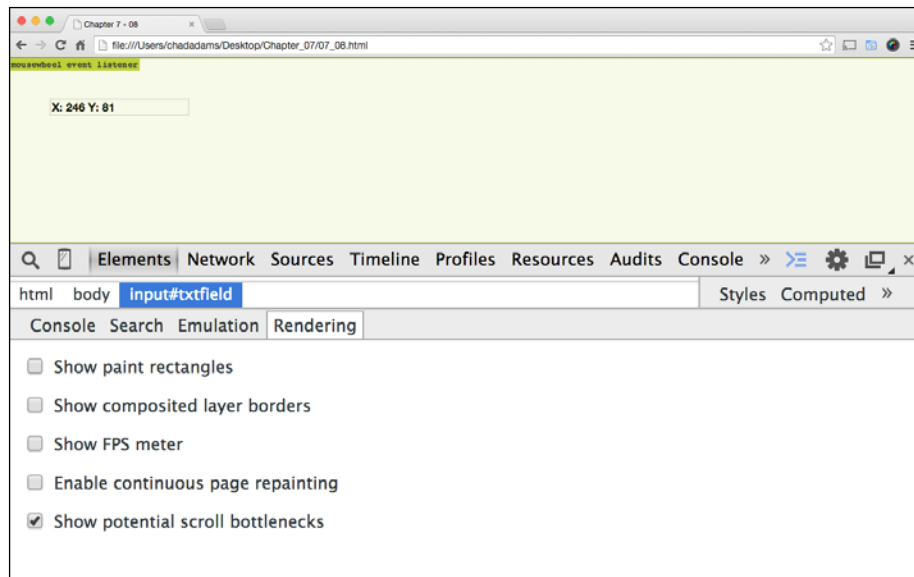
```
1 <!DOCTYPE HTML>
2 <html>
3   <head>
4     <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
5     <title>Chapter 7 - 07</title>
6
7     <style type="text/css">
8       body { font-family: sans-serif; }
9       #txtfield {
10         font-size: 1em;
11         font-weight: bold;
12         margin: 50px;
13       }
14     </style>
15   </head>
16   <body>
17     <input id="txtfield" type="text" value="" />
18   </body>
19   <script type="text/javascript">
20     var txtfield = document.getElementById("txtfield");
21
22     (function(){
23       document.addEventListener("mousewheel", getMouseLocation, false);
24     })()
25
26     function getMouseLocation(event) {
27       txtfield.value = "X: " + event.pageX + " Y: " + event.pageY;
28     }
29   </script>
30 </html>
```

We can see here that the page itself is pretty light but, on line 23, we can see the `mousewheel` event listener in play adding a continuous event using the `getMouseLocation` function on line 26. Then on line 27, our input field with an `id` of `txtfield` is assigned a string with the mouse event information, grabbing the mouse pointer's X and Y coordinates and applying it to the value of `txtfield`. Now let's see **Developer tools** highlight performance issues with scrolling.

Open up the drawer, open the **Rendering** tab, and then click **Show potential scroll bottlenecks**. This will highlight the block areas that have scroll events assigned in JavaScript; here's what our example looks like with the filter enabled:



Now, this by itself isn't too bad when it comes to performance, but applications with multiple mouse movement events can potentially cause issues, even more so if the movement areas overlap. If we add the same event listener to the text area and remove the listener from the document, will we see multiple instances of the scroll listener showing in our **Developer tools** filter? Let's find it out by looking at the output of the final example file for this chapter, 07_08.html:



Nope! As we can see, even when a `mousewheel` event is enabled on a single element, the entire page becomes highlighted. As the `mousewheel` event can be checked at the top of the DOM, the whole page is affected even if an application focuses only on one small element for a `mousewheel` event.

So it's important to keep in mind `mousewheel` events, as they can potentially slow down your page's performance.

Summary

In this chapter, we covered how JavaScript can affect the DOM's performance; we reviewed the `createElement` function and learned how to better write our JavaScript to optimize generating elements from code.

We also reviewed JavaScript animations, and compared their performance to modern CSS3 animations. We also learned how to optimize existing or legacy JavaScript animations.

Lastly, we reviewed paint events in the DOM and saw how the DOM redraws its content after JavaScript manipulates it; we also covered `mousewheel` events and saw how they can potentially slow down the DOM.

In the next chapter, we will take a look at JavaScript's new best friend for performance: *web workers*, and how we can make JavaScript perform like a multithreaded application.

8

Web Workers and Promises

In the previous chapters, we addressed some common performance issues while dealing with common JavaScript issues that come up in general JavaScript development. Now, we come to the point where, assuming that our projects can support newer JavaScript features, we can make our code perform even better than before.

This is where web workers and promises come into play. In this chapter, we will take a look at both and see how and when to use them. We will also discover their limitations and understand their benefits in terms of high-performance JavaScript.

Understanding the limitations first

Before diving into web workers and promises, we need to understand a concern with the JavaScript language itself. As mentioned in past chapters, JavaScript is single-threaded and cannot support two or more methods running at the same time.

For many years, we as JavaScript developers never really had to concern ourselves with threading, let alone the JavaScript memory issues that we have covered in the course of this book. Most of our code existed inside a browser, and ran on the same page either inline, or linked externally to a file on the same server, for basic web page functionality.

As the Web moves forward, with originally frontend coding becoming more and more necessary for high-performing applications, newer ways of handling larger JavaScript applications are needed. Today, we consider these newer features as a part of the *ECMAScript 5* feature set.

In ECMAScript 5, many of these features were rolled into what many consider to be the HTML5 stack. This stack consists of HTML5 (the `DOCTYPE` and `HTML` tags), CSS version 3.0, and ECMAScript 5.

These technologies make the Web much more powerful than the days of AJAX and XHTML development. The limitation is that these features are cutting-edge and may or may not work with every browser. So, using these newer features usually requires a bit of forethought before their implementation in a project.

We've talked about some of these features since *Chapter 2, Increasing Code Performance with JSLint*, including the `use strict` statement, which forces the browser to throw an error if something in JavaScript isn't strictly written or coded correctly. Now you may ask why we are using the `use strict` statement if it isn't supported in all browsers. The trick with the `use strict` statement is that, when it's coded for an older browser, it shows up as a string and is ignored.

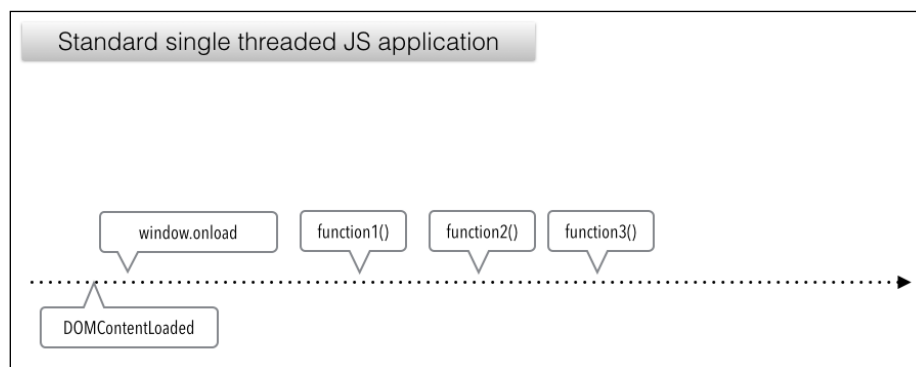
This is a great thing because, even if it is ignored in older browsers, we can still use this newer feature and write more efficient code. Unfortunately, that doesn't translate to all features in ECMAScript 5; this includes web workers and promises.

So in this chapter, let's keep in mind that henceforth, while working with code samples, we need to focus our testing and coding on a newer browser such as Google Chrome, Opera, Firefox, or Apple's Safari, and even newer versions of Internet Explorer that follow the same standards.

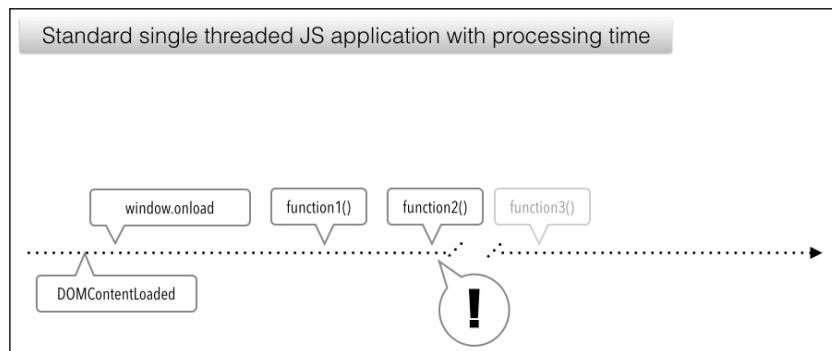
Web workers

Web workers give JavaScript developers like us a way to build multi-threaded JavaScript applications; this works in newer browsers as we have an object called a **worker**. A worker object is simply an external JavaScript file that we pass logic to.

Now, this may seem a little odd. Haven't we worked with external JavaScript files, since the beginning of JavaScript? Fair enough, but web workers are a bit new in terms of how a browser handles the execution of files in the DOM. Let's take a look at the following sample diagram on how a browser reads a file:

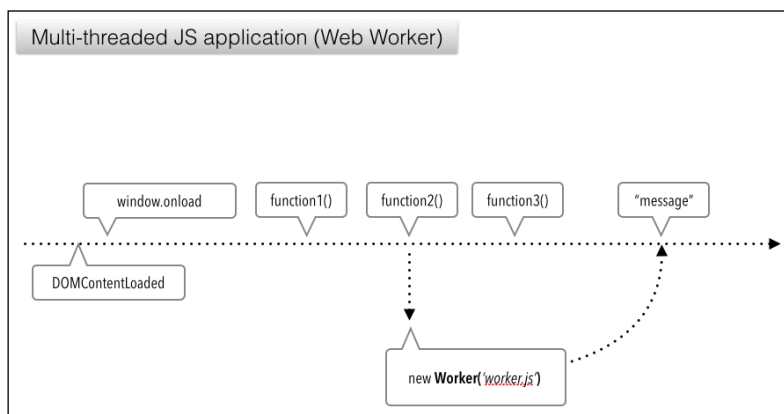


So here we have a single-threaded JavaScript application, a `DOMContentLoaded` event, and a `window.onload` event triggering shortly after, followed by the functions simply named as: `function1`, `function2`, `function3` respectively. Now, what if we had the `function2()` function perform some complex for loop such as calculating pi 5 million times while `console.log(Shakespeare)` is checking the time? Well, we can see that in the following diagram:



As we can see, once the browser calls `function2()`, it locks up and hangs until it can complete its execution, (assuming that the system running the code has enough memory to execute). Now an easy way to fix this would be to say, "Hey, maybe we don't need to check the time, or maybe we only want to calculate pi once to improve performance.". But what if we had no choice but to write code that way? Maybe our application had to work like that, and so we were obliged to code a complicated, slow-performing function that did slow performance; for the success of the application, that function with that logic must fire.

Well, if we must build an application like that, our solution is a web worker. Let's see how that works in comparison to our single-threaded diagrams:



In our example here, we can see in the diagram that we create a new worker that points to an external JavaScript worker file called `worker.js`. That worker sends a response in the form of a message. With web workers, messages are how we pass data between the host script and the worker data. It works in a similar way to any other event in JavaScript using the `onmessage` event.

So how does this look in a coded application? Well, let's find out!

We have a coded example shown in the following screenshot that is built in a similar way to the preceding diagrams:

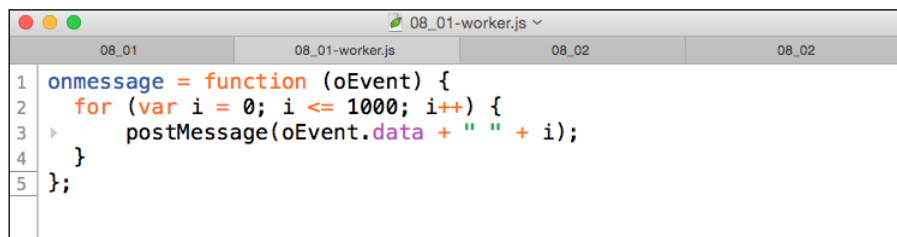


```
1 <!DOCTYPE HTML>
2 <html>
3 <head>
4 <meta charset="utf-8">
5 <title>Chapter 08 - 01</title>
6 </head>
7
8 <body>
9
10
11 <script type="text/javascript">
12
13 function function1() {
14   console.info("function1(): Called.");
15   console.time("Worker");
16 }
17
18 function function2() {
19   var func2_Worker = new Worker("08_01-worker.js");
20   func2_Worker.onmessage = function (oEvent) {
21     console.log("func2_Worker says : " + oEvent.data);
22   };
23
24   func2_Worker.postMessage("Processing a high performance JavaScript worker...");
25 }
26
27 function function3() {
28   console.info("function3(): Called.");
29   console.timeEnd("Worker");
30 }
31
32 window.addEventListener("DOMContentLoaded", function () {
33   console.log("DOM Loaded");
34 }, false);
35 window.addEventListener("load", function () {
36   console.log("Page Loaded");
37
38   function1();
39   function2();
40   function3();
41
42 }, false);
43
44 </script>
45
46 </body>
47 </html>
```

As we can see, this is a simple HTML5 page with a `script` tag on line 11. On line 13, we have `function1()` declared first, which prints an info message to the console; with line 15, we start a new timer to see how fast our worker is. It's rightly called a Worker.

Next up, on line 18 we declare `function2()`; now here is where the things get interesting. First, on line 19, we declare a variable called `func2_worker`; the naming of this variable isn't important, but it's always a good idea to indicate what your variable actually is. In this case, I add the suffix `_Worker` to the variable, following which I create a new web worker using the `Worker` keyword with a capital *W*.

Inside the parentheses, we add a string, the filename, using the relative path of our worker file, named `08_01-worker.js`. Let's take a look inside the worker file.



```
1 onmessage = function (oEvent) {
2   for (var i = 0; i <= 1000; i++) {
3     postMessage(oEvent.data + " " + i);
4   }
5 };
```

As we can see, the worker file is very simple. We have a global object declared on line 1 called `onmessage` and that is assigned a function with a `for` loop. It's also worth noting that we can refer to this context through the `self` and `this` keywords (example: `self.onmessage`). You may have noticed that we also have a parameter called `oEvent`, which is a placeholder for any data being passed into the worker that we call using the `data` property. We can see this in our `postMessage` function on line 3.

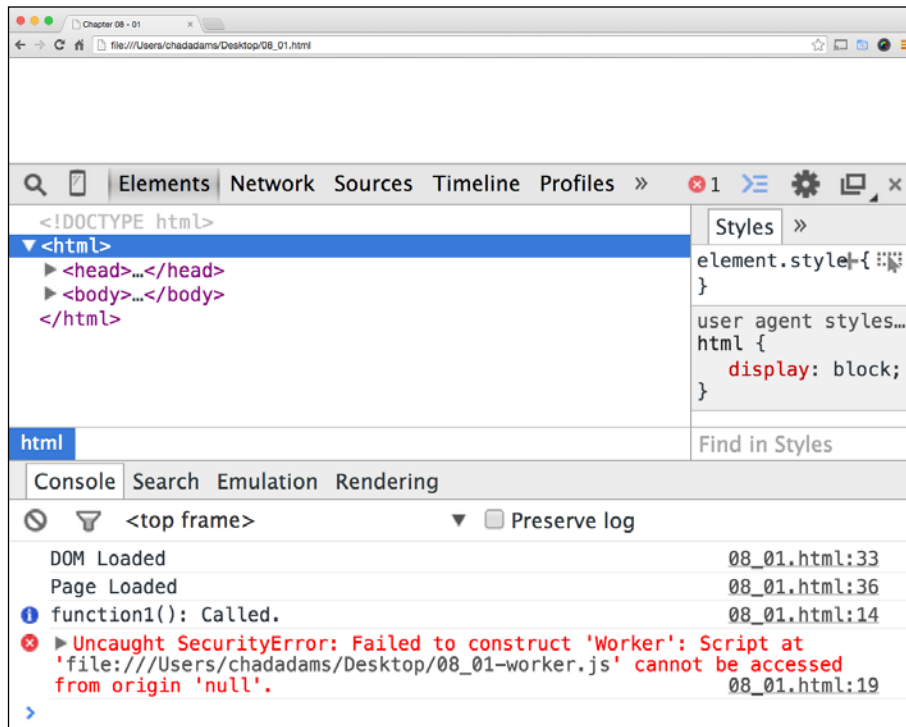
The `postMessage` function is a built-in function for ECMAScript that either sends data to an assigned worker or, if no worker is assigned, it posts a message back to any parent JavaScript workers that might be listening. Now let's go back to our root HTML page script and take a look on line 20; this is shown in the following screenshot:

```
18 function function2() {
19   > var func2_Worker = new Worker("08_01-worker.js");
20   > func2_Worker.onmessage = function (oEvent) {
21     > console.log("func2_Worker says : " + oEvent.data);
22   > };
23   >
24   > func2_Worker.postMessage("Processing a high performance JavaScript worker...");
25 }
26
27 function function3() {
28   > console.info("function3(): Called.");
29   > console.timeEnd("Worker");
30 }
31
32 window.addEventListener("DOMContentLoaded", function () {
33   > console.log("DOM Loaded");
34 }, false);
35 window.addEventListener("load", function () {
36   > console.log("Page Loaded");
37   >
38   > function1();
39   > function2();
40   > function3();
41   >
42 }, false);
43
44 </script>
45 >
46 </body>
47 </html>
```

We can see that, by calling our `func2_Worker` worker variable, we can use the `onmessage` property of that worker and call a function back on our root page; in this case, you need to log a message back to the console using the `oEvent` parameter used in the worker.

That's all well and good. But how do we pass data in? Well, that's easy enough. Line 24 uses the `func2_Worker` variable but utilizes the `postMessage` function for that worker object, as mentioned earlier. As we've assigned a worker variable to this `postMessage` function, this will pass in a data parameter into our `oEvent` parameter used in our `worker.js` file; in this case, it's a string that says, "Processing a high performance JavaScript worker...".

Finally, on line 32 and again on line 35, we have 2 event listeners. One is for the `DOMContentLoaded` event that was shown in our diagram as the first function called in our execution thread and that simply outputs a log message that the DOM is loaded; this is followed by our `window.onload` function, which also prints a log message, But then it also triggers functions 1, 2, and 3, in order when the page is loaded. Let's load this in our browser and see what happens by using Chrome's **Developer tools** option. Take a look at the output in the **Console** panel, which will resemble the following screenshot:



Well, that's not a good sign as we can see an error appear in our console. The DOM Loaded and Page Loaded log messages appear as does `function1(): Called.` after which we get the `Uncaught SecurityError: Failed to construct 'Worker': Script at (file:url) cannot be accessed from origin 'null'` error message.

Now what does that mean? First, we have to understand that using a web worker is similar to working with AJAX. If your code is not on a server, there is a security risk in sharing or gathering data across your system. Now this isn't incorrect but, when testing our code, we need to test on a local server such as Apache or IIS that can secure our content using HTTP. In Chrome, there is also another way to disable this warning, but that's only for limited testing.

Testing workers with a local server

A local server can be created on OS X and Linux quickly using Python; now, if you're not a Python *guru*, don't worry as this is a quick one-line bit of Terminal code to spin up a server in seconds.

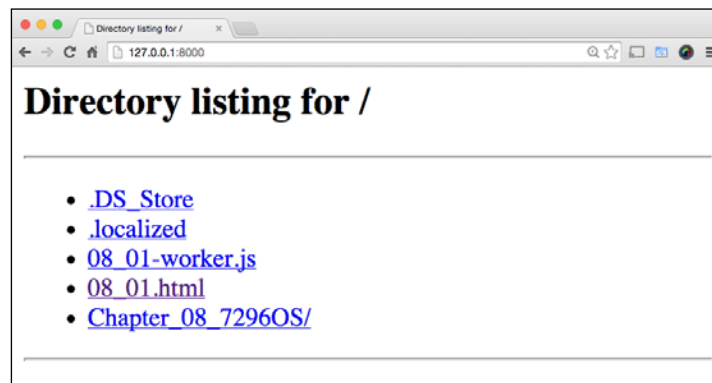
First, open the Terminal and set its path; this should be the path your files are in. You can do this by using the change directory command or `cd`. Here's an example of setting the path to the active user's desktop using the *tilde* key:

```
cd ~/Desktop
```

Once that's done, we can start the server with this simple one-line python command that calls a built-in simple server method, as follows:

```
python -m SimpleHTTPServer
```

Once we press the *Enter* key, we can start the server. We can look at the server root by typing `http://127.0.0.1:8000` into Chrome; we should then see a list of files to access. Also, should you need to close the server, you can exit out of the Terminal or use *CTRL* + *Z* to kill the server manually.



Now go ahead and open the HTML file in which the worker script is called, from the `.js` file, in the page. We should then see the **Console** panel in Chrome's **Developer tools** show one thousand lines iterating through our "for loop" from our worker JavaScript file.

We can also see that, on the fifth console line, the `console.timeEnd` function stops for about 0.5 milliseconds, showing that it's been called prior to processing the loop. This is shown in the following screenshot:

```

<top frame>
DOM Loaded                                08_01.html:33
Page Loaded                              08_01.html:36
function1(): Called.                      08_01.html:14
function3(): Called.                      08_01.html:28
Worker: 0.475ms                           08_01.html:29
func2_Worker says : Processing a high performance JavaScript worker... 0  08_01.html:21
func2_Worker says : Processing a high performance JavaScript worker... 1  08_01.html:21
func2_Worker says : Processing a high performance JavaScript worker... 2  08_01.html:21
func2_Worker says : Processing a high performance JavaScript worker... 3  08_01.html:21
func2_Worker says : Processing a high performance JavaScript worker... 4  08_01.html:21
func2_Worker says : Processing a high performance JavaScript worker... 5  08_01.html:21
func2_Worker says : Processing a high performance JavaScript worker... 6  08_01.html:21
func2_Worker says : Processing a high performance JavaScript worker... 7  08_01.html:21
func2_Worker says : Processing a high performance JavaScript worker... 8  08_01.html:21
func2_Worker says : Processing a high performance JavaScript worker... 9  08_01.html:21
func2_Worker says : Processing a high performance JavaScript worker... 10 08_01.html:21

```

Before we move on, let's check how long this would process outside a worker in the next code sample. We've recreated the logic for the loop in the page itself without using a web worker. We are still using the `console.time` function to test how long the thread runs until `function3()` is triggered. Let's take a look at the following code and review it:

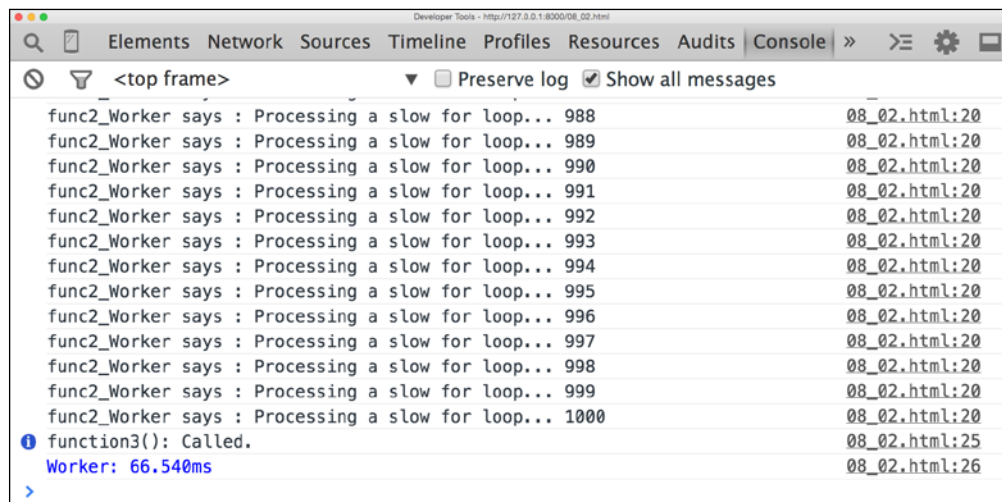
```

1 <!DOCTYPE HTML>
2 <html>
3 <head>
4 <meta charset="utf-8">
5 <title>Chapter 08 - 02</title>
6 </head>
7
8 <body>
9
10
11 <script type="text/javascript">
12
13 function function1() {
14   console.info("function1(): Called.");
15   console.time("Worker");
16 }
17
18 function function2() {
19   for (var i = 0; i <= 1000; i++) {
20     console.log("func2_Worker says : Processing a slow for loop... " + i);
21   }
22 }
23
24 function function3() {
25   console.info("function3(): Called.");
26   console.timeEnd("Worker");
27 }
28
29 window.addEventListener("DOMContentLoaded", function () {
30   console.log("DOM Loaded");
31 }, false);
32 window.addEventListener("load", function () {
33   console.log("Page Loaded");
34
35   function1();
36   function2();
37   function3();
38
39 }, false);
40
41 </script>
42
43 </body>
44 </html>

```


So, on line 19, we've removed the reference to the worker file, which is a `.js` file, and moved the `for` loop into the page. Here it will loop one thousand times and print to the console. Now on line 32, we have our `window.load` event listener, and we sequentially call our functions as 1, 2, and 3.

We then again use the `console.time` function to track how long a process occurs. As this code sample is now single-threaded, we should see a longer time for the `timeEnd` function to fire. Let's run our code and take a look at the next screenshot:



That's not bad! Here, our time is much longer than our multi-threaded Worker example, which is roughly 70 milliseconds slower than our web worker. That's not a bad performance boost; it's minor but still helpful. Now, one issue with workers is that they take a lot of time to trigger the next function that exists on a separate thread from the main thread. We need to have some way to call a function when a function completes asynchronously, and for that we have JavaScript promises.

Promises

JavaScript promises are also a new way to optimize our JavaScript code; the idea of promises is you have a function that is chained to a main function, and is fired in sequential order as it's written. Here is how it's structured. First, we create a new object using the `Promise` object, and, inside the parentheses, we write the main function, and assign the new promise object to a variable.

One thing to note before continuing is that JavaScript promises are EcmaScript 6- specific. So, for this section, we will need to test our code in an EcmaScript 6-ready browser such as Google Chrome.

Next, with our `promise` variable, we use the `then` keyword, which in actuality works just like a function. But it only fires when our root promise function is completed. Also, we can chain the `then` keywords one after another and sequentially fire JavaScript asynchronously to ensure that the variables scoped in our promise will, of course, promise to the next `then` function that those variables will have set values. Let's take a look at a sample promise and see how this works:

```

1 <!DOCTYPE HTML>
2 <html>
3 <head>
4 <meta charset="utf-8">
5 <title>Chapter 08 - 03</title>
6 <style type="text/css">
7   body { font-family: monospace; font-size: 1em; }
8   #results { margin-top: 1em; }
9 </style>
10 </head>
11
12 <body>
13 <button onclick="makeAPromise()">Make A Promise</button>
14
15 <div id="results"></div>
16
17 <script type="text/javascript">
18
19   function makeAPromise() {
20     var promiseCount = 0;
21
22     var promiseNo1 = new Promise(
23     function(resolve) {
24       for (var i = 0; i <= 1000; i++) {
25         promiseCount = i * 5;
26       }
27
28       /* Assign a value to our promise */
29       resolve("Our final count: <strong>" + promiseCount + "</strong>")
30     });
31
32     promiseNo1.then(
33     function(promiseCount) {
34       document.getElementById("results").innerHTML = promiseCount;
35     }
36     )
37   }
38
39 </script>
40 </body>
41 </html>

```

In our code sample, we have an HTML5 page with an embedded `script` tag. We have two elements on our page that we interact with or view using a `button` tag with the `makeAPromise()` function as an attached `onclick` event on line 13. On line 15, we have a `div` tag with an `id` of `results`, with its inner HTML left empty.

Next, on line 19 we create our `makeAPromise` function and set a `count` variable on line number 20 called `promiseCount`. Now here's where we create our promise. On line number 22, we create a variable called `promiseNo1`, and we assign it with a new `Promise` object. Here you can notice how we start opening the parenthesis with a function as parameter that starts on line 23, and we have a `resolve` parameter inside that function. We'll discuss that later.

In our `Promise` function, we have a simple `for` loop that multiplies the value from the `for` loop by 5, and the `then` function assigns it to our `promiseCount` variable. To finish our `Promise` object's function, notice a new type of keyword, `resolve`! The `resolve` keyword is a type of return used specifically for promises; it sets a promise's return value. There are also other promise return types such as `reject`, which allows us to return a failed value if we want. For this example, however, we are keeping it simple and only using `resolve`.

Now, if you remember on line 23, our `Promise` function had an internal function with the `resolve` parameter. Though this may look a little odd, it is required to make our promise work; by adding `resolve` to our function, we are telling our promise that we need to use the `resolve` function inside our `Promise` function. For example, if we needed `resolve` and `reject`, we would write it as `function (resolve, reject) {}`.

Back on line 29, we assign our `resolve` with a string that outputs our value with some copy to fill our `div`, but we don't assign the `innerHTML` property here; that's done using our `promiseNo1.then` function. This works like a function that follows the promise's `resolve` function.

Finally on line 32, we call our `promiseNo1` variable's instance, use the `then` function, and again wrap the parenthesis with its own internal function. We may notice that, on line 33, we've passed in an argument called `promiseCount`. This is our `resolve` value that ended our `Promise` function declared on line 22. We then use that again in line 33, where we assign our `results` `div` element with its `innerHTML` property.

Testing a true asynchronous promise

For this simple example, we can see how a promise is structured and how each firing is needed when chained; when we chain promises, we can see how a promise can still fire a chained function even when we create a bit of single-threaded JavaScript code that causes a delay in execution. In this case, it's a `setTimeout` function; let's take a look at this new code sample shown in the following screenshot:

```

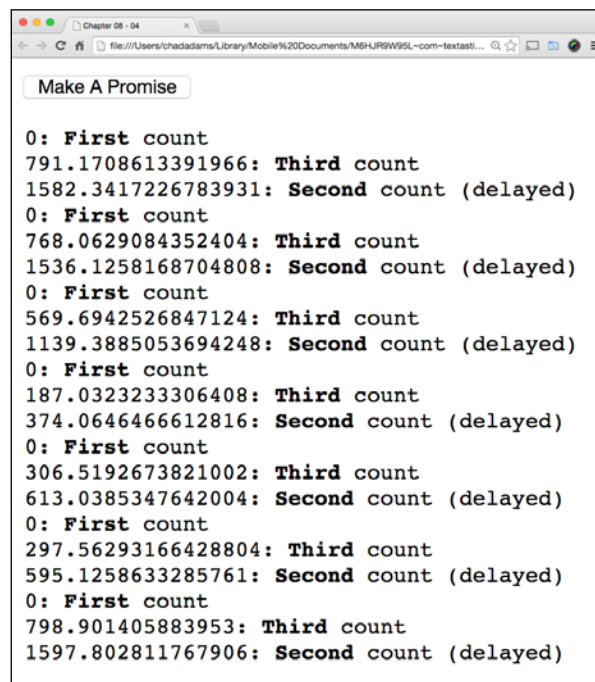
1 <!DOCTYPE HTML>
2 <html>
3 <head>
4 <meta charset="utf-8">
5 <title>Chapter 08 - 04</title>
6 <style type="text/css">
7   body { font-family: monospace; font-size: 1em; }
8   #results { margin-top: 1em; }
9 </style>
10 </head>
11
12 <body>
13 <button onclick="makeAPromise()">Make A Promise</button>
14
15 <div id="results"></div>
16
17 <script type="text/javascript">
18
19   function makeAPromise() {
20     var timerCount = 0;
21     document.getElementById("results").innerHTML +=
22     timerCount + ": <strong>First</strong> count <br />";
23
24     var promiseNo1 = new Promise(
25     function(resolve) {
26       for (var i = 0; i <= 1000; i++) {
27         timerCount = i * Math.random();
28       }
29       resolve(timerCount);
30     });
31     promiseNo1.then(
32     function(response) {
33       var totalCount = response + timerCount;
34       window.setTimeout(function(){
35         document.getElementById("results").innerHTML +=
36         totalCount +
37         ": <strong>Second</strong> count (delayed)<br />";
38       }, timerCount);
39     })
40     .then(
41     function() {
42       document.getElementById("results").innerHTML +=
43       timerCount + ": <strong>Third</strong> count <br />";
44     })
45     );
46   }
47
48 </script>
49 </body>
50 </html>

```

For this simple example, we can see how promise chains function without breaking the thread. Here, we set a `timerCount` variable on line 20; then we will print to an empty `results` `div` element found on line 15. Next, by reusing our `promiseNo1` variable with its own promise instance, we create a `for` loop that randomizes the `timerCount` using `Math.random()`, which allows a random number to be generated and then multiplied to 10000 in the `for` loop when it finishes.

Lastly, we use the `resolve` function to return our promise, which is chained to our `then` function on line 31; here we have an argument called `response` to serve as our `resolve` function's value. Now on line 33, we have a variable called `totalCount`, where we have the `response` argument and the `timerCount` function added together.

Next, we create a `setTimeout` function that appends the `results div` element with a second line, printing the amount of time set by the `totalCount` variable declared by us, while still using the `timerCount` function as our `timeout` value. Now, the last part of our chain is another `then` function on line number 40. Here, we append the `results div` element again, but you need to note that we are printing from our second chained `then` function. Let's take a look at how that works in Chrome, as shown in the following screenshot:



Take a look at the output. Here, we can see that, with each button click, we get a numeric count for each point of the promise chain. We have a value of 0 on the `First count`, and a random larger number on the `Third count`. Wait! Is that the third count? Yes, note that the third count comes after first; this demonstrates that, even when we were waiting for our `for` loop to process, the third promise continued.

On the next line, we see an even larger number value with the `Second count` noted in our line; if we continue clicking the button, we should see a consistent pattern. Using promises can help us multi-thread code as long as we don't need a specific value in the chain immediately. We also get performance benefits by moving some of our code off our main JavaScript thread using promises.

Summary

In this chapter, we reviewed how to use web workers, and the limitations web workers have technically and conceptually in a real-life application. We also worked with JavaScript promises, where we learned common keywords related to promises such as `respond` and `revoke`. We saw how to use the `then` function to chain our promises in sync with our main application thread, to create a multi-threaded JavaScript function.

In the next chapter, we will see how working from a mobile device such as iOS and Android can affect our performance and how to debug performance on a device.

9

Optimizing JavaScript for iOS Hybrid Apps

In this chapter, we are going to take a look at the process of optimizing JavaScript for iOS web apps (also known as hybrid apps). We will take a look at some common ways of debugging and optimizing JavaScript and page performance, both in a device's web browser and a standalone app's web view.

Also, we'll take a look at the Apple Web Inspector and see how to use it for iOS development. Finally, we will also gain a bit of understanding about building a hybrid app and learn the tools that help to better build JavaScript-focused apps for iOS. Moreover, we'll learn about a class that might help us further in this.

We are going to learn about the following topics in the chapter:

- Getting ready for iOS development
- iOS hybrid development

Getting ready for iOS development

Before starting this chapter with Xcode examples and using iOS Simulator in a JavaScript performance book, I will be displaying some native code and will use tools that haven't been covered in this course. Mobile app developments, regardless of platform, are books within themselves. When covering the build of the iOS project, I will be briefly going over the process of setting up a project and writing *non-JavaScript* code to get our JavaScript files into a hybrid iOS WebView for development. This is essential due to the way iOS secures its HTML5-based apps. Apps on iOS that use HTML5 can be debugged, either from a server or from an app directly, as long as the app's project is built and deployed in its debug setting on a host system (meaning the developers machine).


Readers of this book are not expected to know how to build a native app from the beginning to the end. And that's completely acceptable, as you can copy-and-paste, and follow along as I go. But I will show you the code to get us to the point of testing JavaScript code, and the code used will be the smallest and the fastest possible to render your content.

All of these code samples will be hosted as an Xcode project solution of some type on Packt Publishing's website, but they will also be shown here if you want to follow along, without relying on code samples. Now with that said, lets get started...

iOS hybrid development

Xcode is the IDE provided by Apple to develop apps for both iOS devices and desktop devices for Macintosh systems. As a JavaScript editor, it has pretty basic functions, but Xcode should be mainly used in addition to a project's toolset for JavaScript developers. It provides basic code hinting for JavaScript, HTML, and CSS, but not more than that.

To install Xcode, we will need to start the installation process from the Mac App Store. Apple, in recent years, has moved its IDE to the Mac App Store for faster updates to developers and subsequently app updates for iOS and Mac applications. Installation is easy; simply log in with your Apple ID in the Mac App Store and download Xcode; you can search for it at the top or, if you look in the right rail among popular free downloads, you can find a link to the Xcode Mac App Store page. Once you reach this, click **Install** as shown in the following screenshot:



Xcode

Create great apps for Mac, iPhone, and iPad.

Installing ▾

Xcode 4+

Essentials

Xcode provides everything developers need to create great applications for Mac, iPhone, and iPad. Xcode brings user interface design, coding, testing, and debugging all into a unified workflow. The Xcode IDE combined with the Cocoa and Cocoa Touch frameworks, and the Swift programming language make developing apps easier and more fun than ever before.

...More

What's New in Version 6.1.1

Includes SDKs for OS X 10.10 Yosemite, OS X 10.9 Mavericks, and iOS 8.1

...More

Apple Web Site >

Xcode Support >

App License Agreement >

Privacy Policy >

Information

Category: Developer Tools

Updated: Dec 02, 2014

Version: 6.1.1

Price: Free

Size: 2.49 GB

Family Sharing: Yes

Language: English

Seller: Apple Inc.

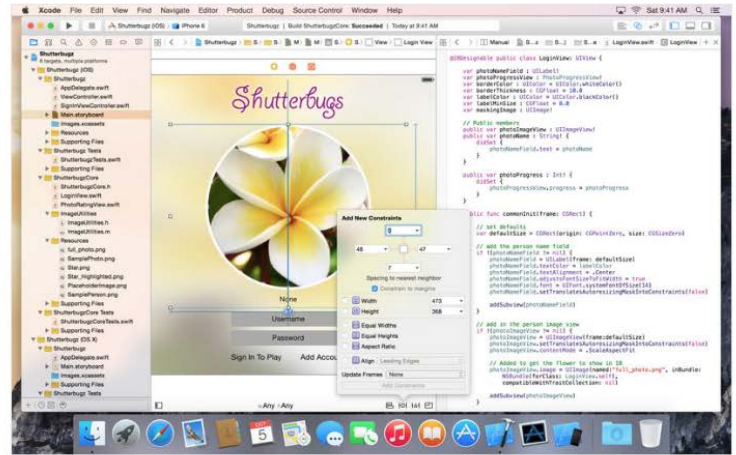
© 1999–2014 Apple Inc.

Rated 4+

Compatibility: OS X 10.9.4 or later

More by Apple

- OS X Yosemite Utilities ★★★★★
- iPhoto Photography ★★★★★
- iMovie Video ★★★★★
- Pages Productivity ★★★★★

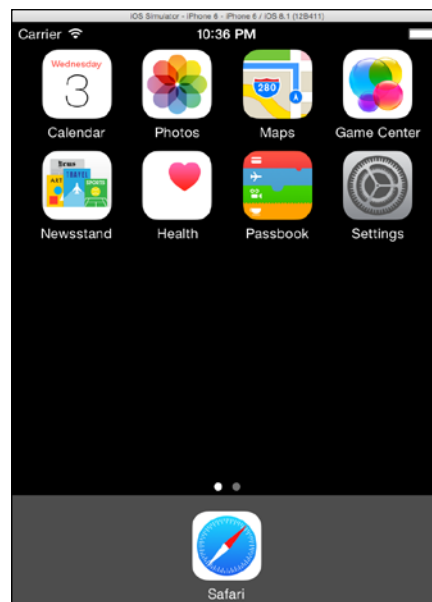


It's important to know that, for the sake of simplicity in this chapter, we will not deploy an app to a device; so if you are curious about it, you will need to be actively enrolled in Apple's Developer Program. The cost is 99 dollars a year, or 299 dollars for an enterprise license that allows deployment of an app outside the control of the iOS App Store.

If you're curious to learn more about deploying to a device, the code in this chapter will run on the device assuming that your certificates are set up on your end.

For more information on this, check out Apple's iOS Developer Center documentation online at https://developer.apple.com/library/ios/documentation/IDEs/Conceptual/AppDistributionGuide/Introduction/Introduction.html#//apple_ref/doc/uid/TP40012582.

Once it's installed, we can open up Xcode and look at the iOS Simulator; we can do this by clicking **XCode**, followed by **Open Developer Tool**, and then clicking on **iOS Simulator**. Upon first opening iOS Simulator, we will see what appears to be a simulation of an iOS device, shown in the next screenshot. Note that this is a simulation, *not* a real iOS device (even if it feels pretty close).



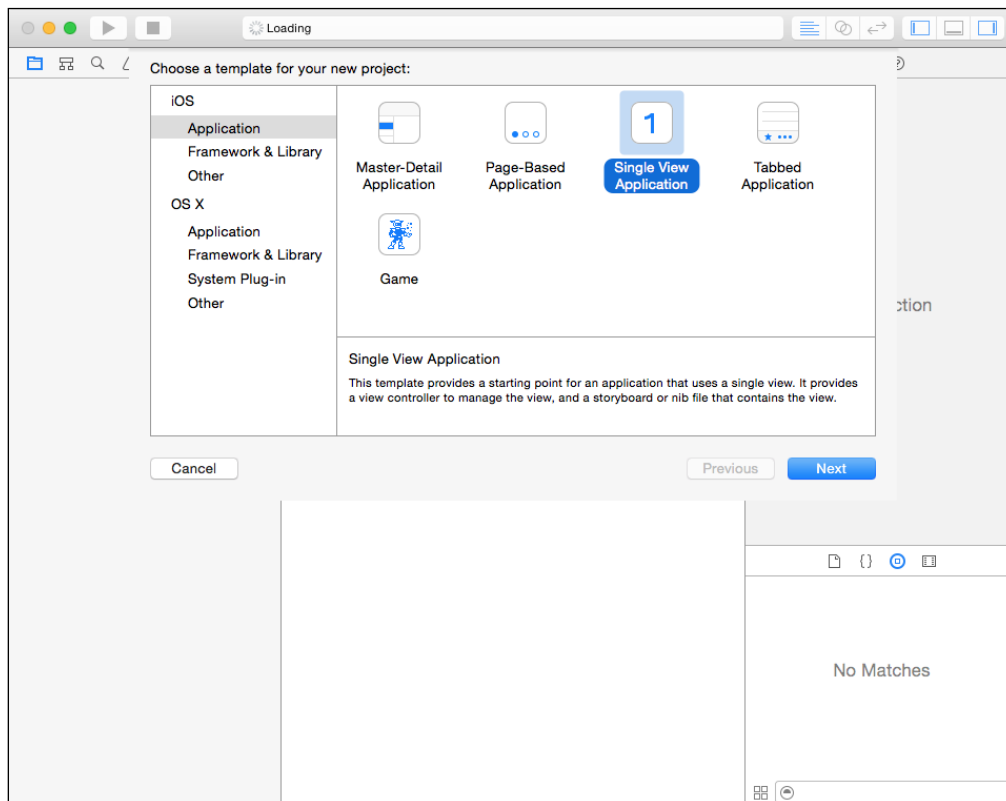
A neat trick for JavaScript developers working with local HTML files outside an app is that they can quickly drag-and-drop an HTML file. Due to this, the simulator will open the mobile version of Safari, the built-in browser for iPhone and iPads, and render the page as it would do on an iOS device; this is pretty helpful when testing pages before deploying them to a web server.

Setting up a simple iOS hybrid app

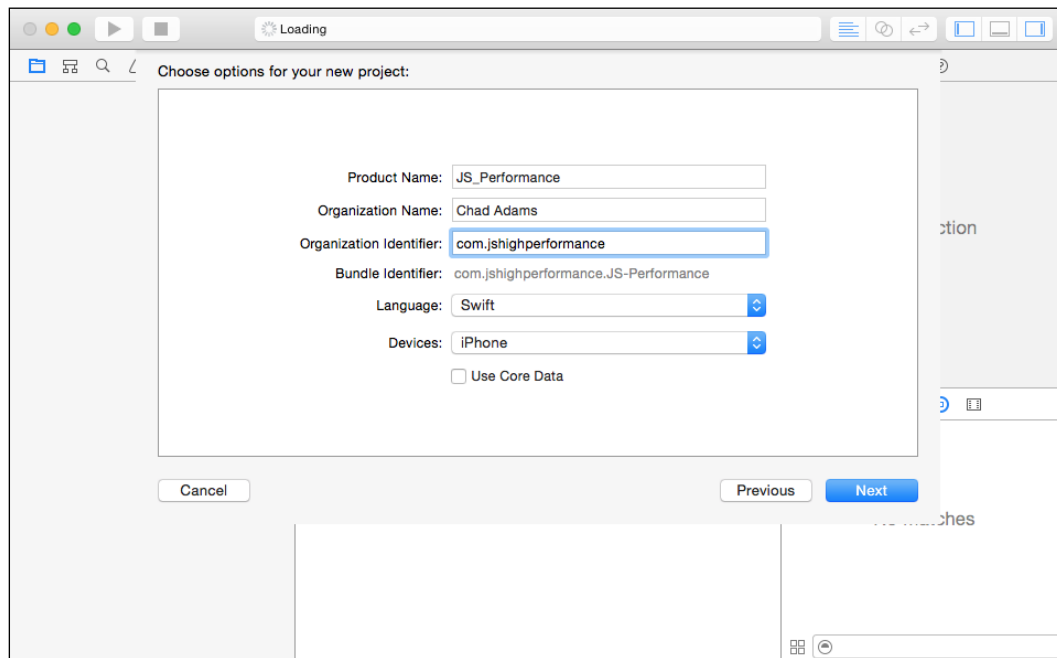
JavaScript performance on a built-in hybrid application can be much slower than the same page run on the mobile version of Safari. To test this, we are going to build a very simple web browser using Apple's new programming language **Swift**. Swift is an iOS-ready language that JavaScript developers should feel at home with.

Swift itself follows a syntax similar to JavaScript but, unlike JavaScript, variables and objects can be given types allowing for stronger, more accurate coding. In that regard, Swift follows syntax similar to what can be seen in the *ECMAScript 6* and *TypeScript* styles of coding practice. If you are checking these newer languages out, I encourage you to check out Swift as well.

Now let's create a simple web view, also known as a **UIWebView**, which is the class used to create a web view in an iOS app. First, let's create a new iPhone project; we are using an iPhone to keep our app simple. Open Xcode and select the **Create new Xcode project** project; then, as shown in the following screenshot, select the **Single View Application** option and click the **Next** button.

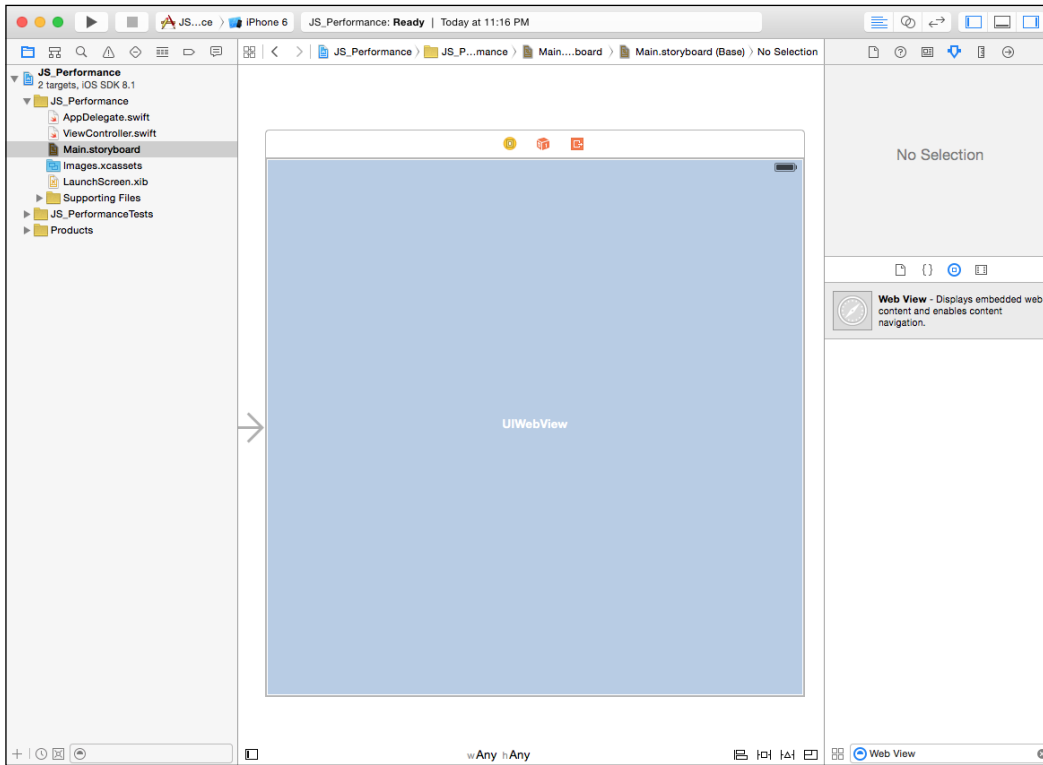


On the next view of the wizard, set the product name as `JS_Performance`, the language to **Swift**, and the device to **iPhone**; the organization name should autofill with your name based on your account name in the OS. The organization identifier is a reverse domain name unique identifier for our app; this can be whatever you deem appropriate. For instructional purposes, here's my setup:



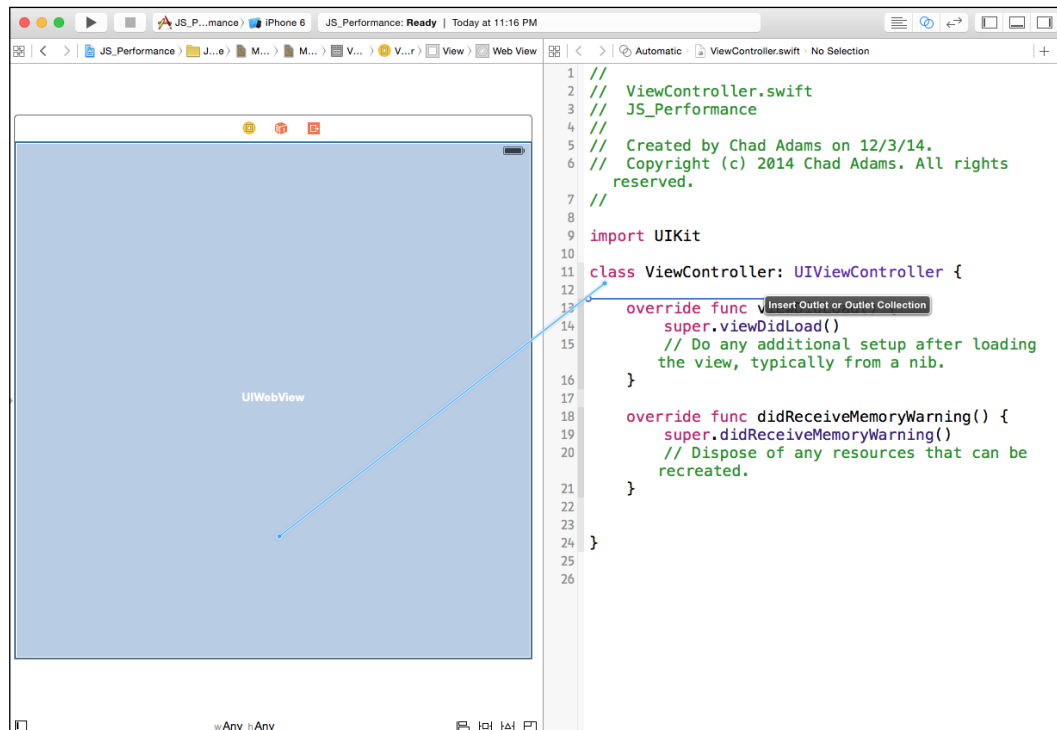
Once your project names are set, click the **Next** button and save to a folder of your choice with **Git repository** left unchecked. When that's done, select **Main.storyboard** under your **Project Navigator**, which is found in the left panel. We should be in the storyboard view now. Let's open the **Object Library**, which can be found in the lower-right panel in the subtab with an icon of a square inside a circle.

Search for **Web View** in the **Object Library** in the bottom-right search bar, and then drag that to the square view that represents our iOS view.



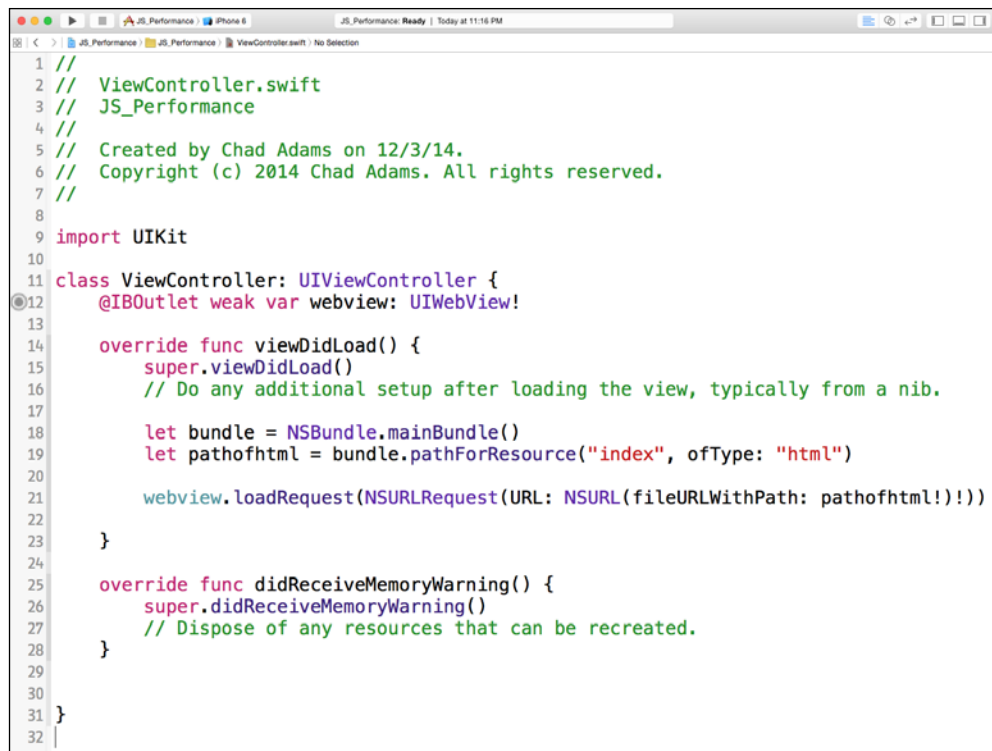
We need to consider two more things before we link up an HTML page using Swift; we need to set constraints as native iOS objects will be stretched to fit various iOS device windows. To fill the space, you can add the constraints by selecting the **UIWebView** object and pressing *Command + Option + Shift + =* on your Mac keyboard. Now you should see a blue border appear briefly around your UIWebView.

Lastly, we need to connect our **UIWebView** to our Swift code; for this, we need to open the **Assistant Editor** by pressing *Command + Option + Return* on the keyboard. We should see **ViewController.swift** open up in a side panel next to our **Storyboard**. To link this as a code variable, right-click (or option-click the **UIWebView** object) and, with the button held down, drag the **UIWebView** to line number 12 in the **ViewController.swift** code in our **Assistant Editor**. This is shown in the following diagram:



Once that's done, a popup will appear. Now leave everything the same as it comes up, but set the name to `webview`; this will be the variable referencing our **UIWebView**. With that done, save your `Main.storyboard` file and navigate to your `ViewController.swift` file.

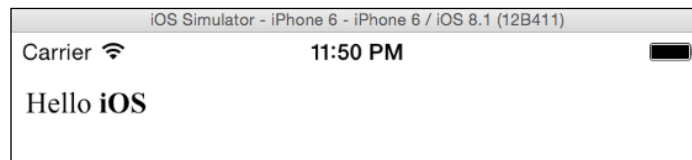
Now take a look at the Swift code shown in the following screenshot, and copy it into the project; the important part is on line 19, which contains the filename and type loaded into the web view; which in this case, this is `index.html`.



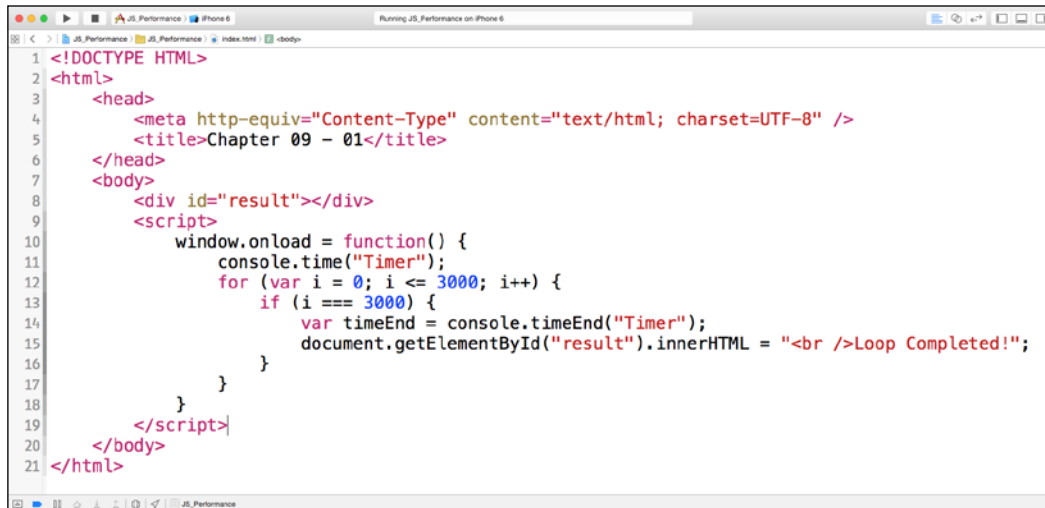
Now obviously, we don't have an `index.html` file, so let's create one. Go to **File** and then select **New** followed by the **New File** option. Next, under **iOS** select **Empty Application** and click **Next** to complete the wizard. Save the file as `index.html` and click **Create**. Now open the `index.html` file, and type the following code into the HTML page:

```
<br />Hello <strong>iOS</strong>
```

Now click **Run** (the play button in the main iOS task bar), and we should see our HTML page running inside our own app, as shown here:



That's nice work! We built an iOS app with Swift (even if it's a simple app). Let's create a structured HTML page; we will override our `Hello iOS` text with the HTML shown in the following screenshot:

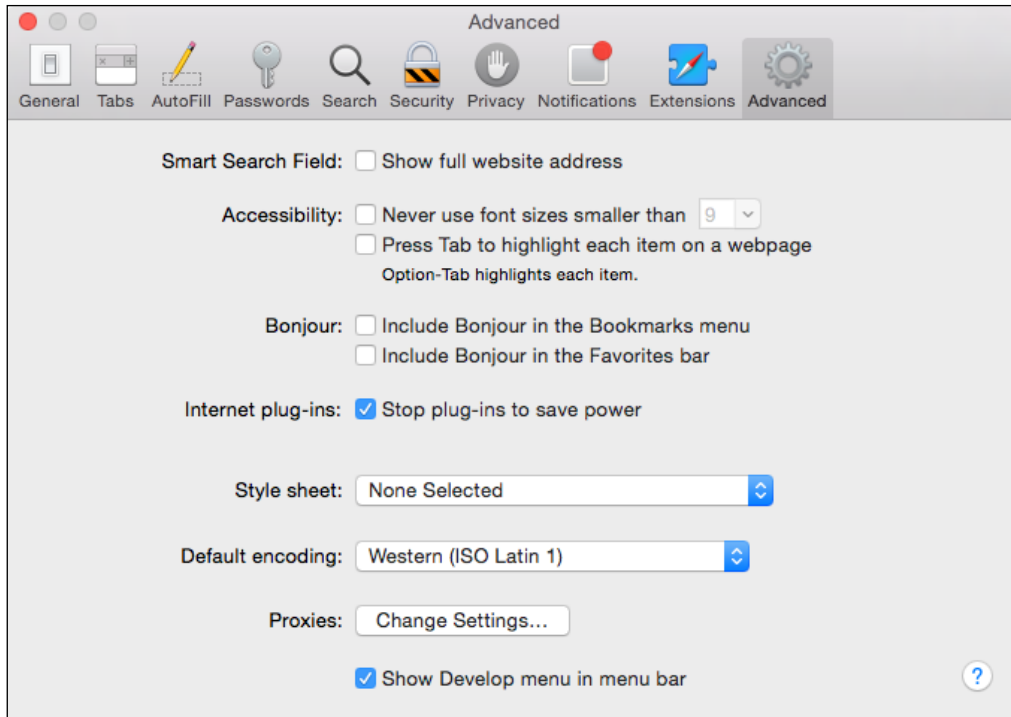


Here, we use the standard `console.time` function and print a message to our `UIWebView` page when finished; if we hit **Run** in Xcode, we will see the `Loop Completed` message on load. But how do we get our performance information? How can we get our `console.timeEnd` function code on line 14 on our HTML page?

Using Safari Web Inspector for JavaScript performance

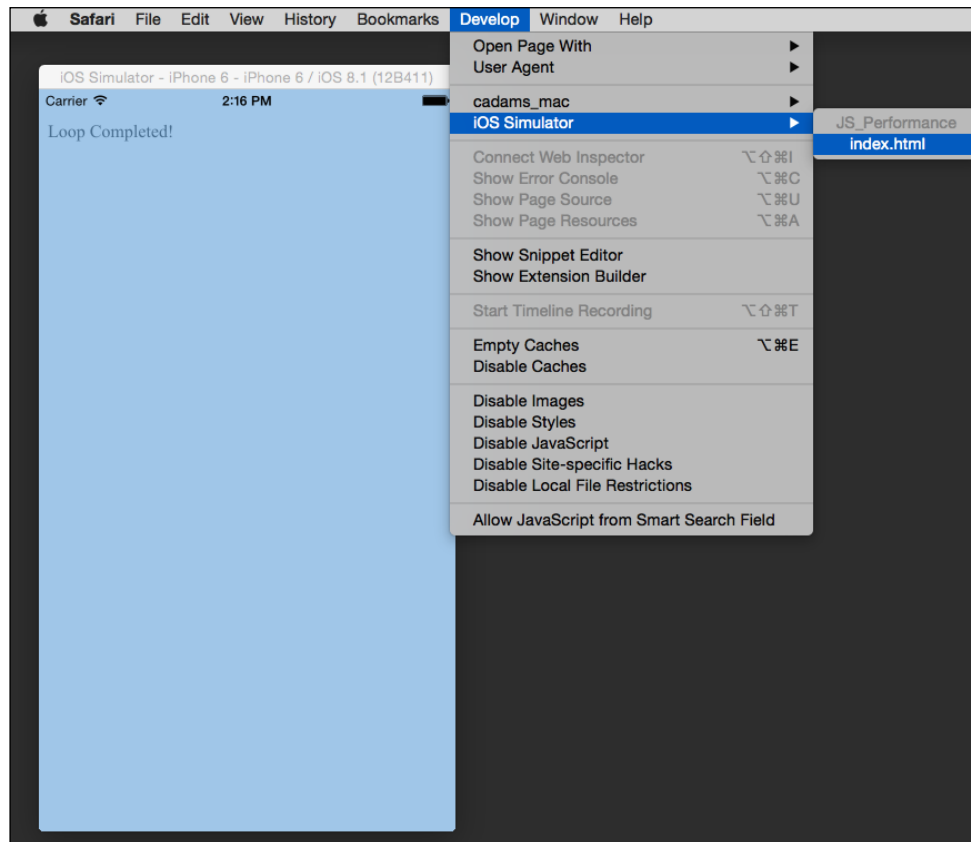
Apple does provide a Web Inspector for `UIWebViews`, and it's the same inspector for desktop Safari. It's easy to use, but has an issue: the inspector only works on iOS Simulators and devices that have started from an Xcode project. This limitation is due to security concerns for hybrid apps that may contain sensitive JavaScript code that could be exploited if visible.

Let's check our project's embedded HTML page console. First, open desktop Safari on your Mac and enable developer mode. Launch the **Preferences** option. Under the **Advanced** tab, ensure that the **Show develop menu in menu bar** option is checked, as shown in the following screenshot:



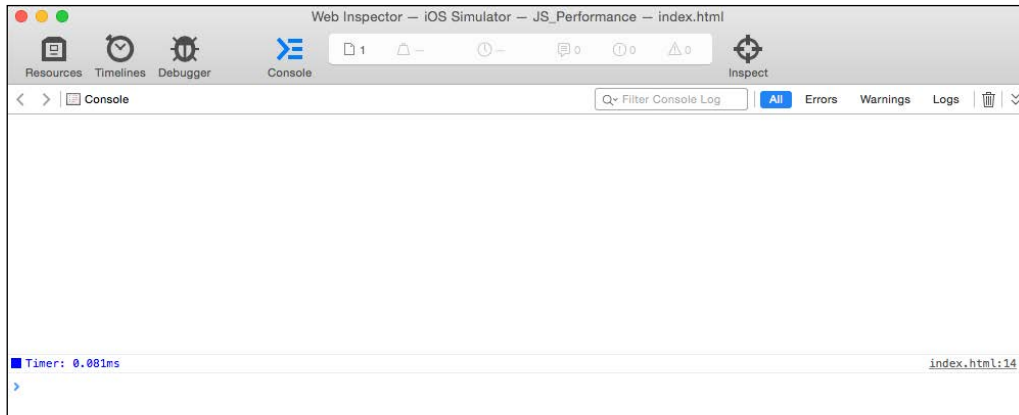
Next, let's rerun our Xcode project, start up iOS Simulator and then rerun our page. Once our app is running with the **Loop Completed** result showing, open desktop Safari and click **Develop**, then **iOS Simulator**, followed by **index.html**.

If you look closely, you will see iOS simulator's UIWebView highlighted in blue when you place the mouse over **index.html**; a visible page is seen as shown in the following screenshot:



Once we release the mouse on **index.html**, we Safari's **Web Inspector** window appears featuring our hybrid iOS app's DOM and console information. The Safari's **Web Inspector** is pretty similar to Chrome's **Developer tools** in terms of feature sets; the panels used in the **Developer tools** also exist as icons in **Web Inspector**.

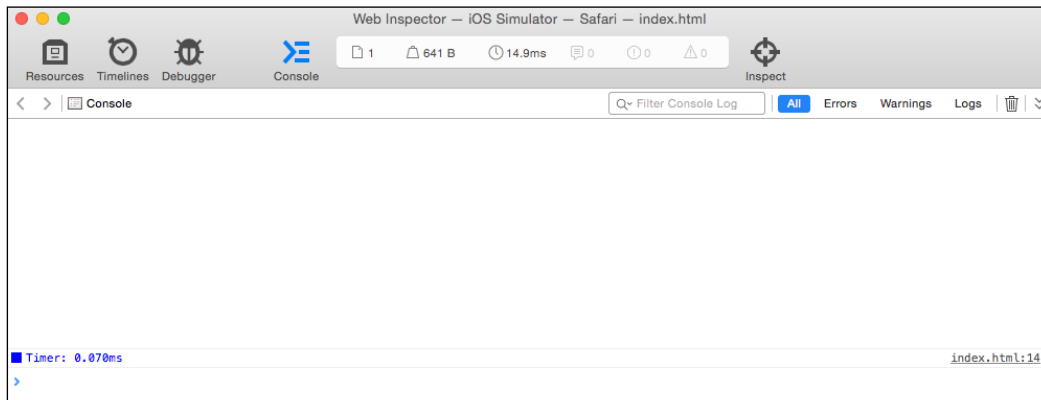
Now let's select the **Console** panel in **Web Inspector**. Here, we can see our full console window including our `Timer.console.time` function test included in the `for` loop. As we can see in the following screenshot, the loop took 0.081 milliseconds to process inside iOS.



Comparing UIWebView with Mobile Safari

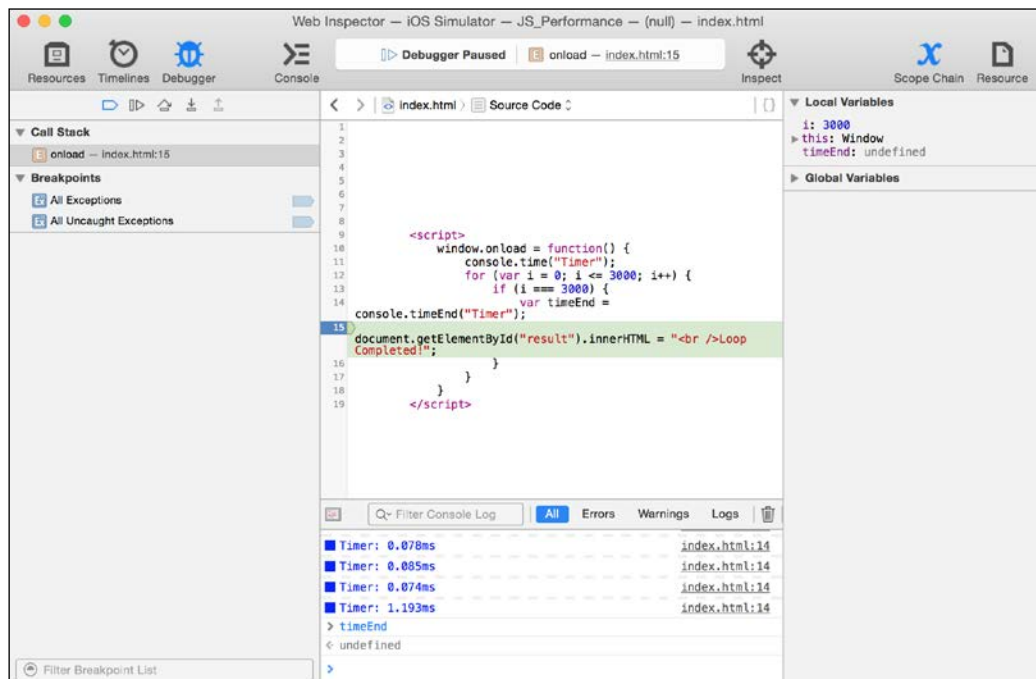
What if we wanted to take our code and move it to Mobile Safari to test? This is easy enough; as mentioned earlier in the chapter, we can drag-and-drop the `index.html` file into our iOS Simulator, and then the OS will open the mobile version of Safari and load the page for us.

With that ready, we will need to reconnect Safari **Web Inspector** to the **iOS Simulator** and reload the page. Once that's done, we can see that our `console.time` function is a bit faster; this time it's roughly 0.07 milliseconds, which is a full .01 milliseconds faster than UIWebView, as shown here:



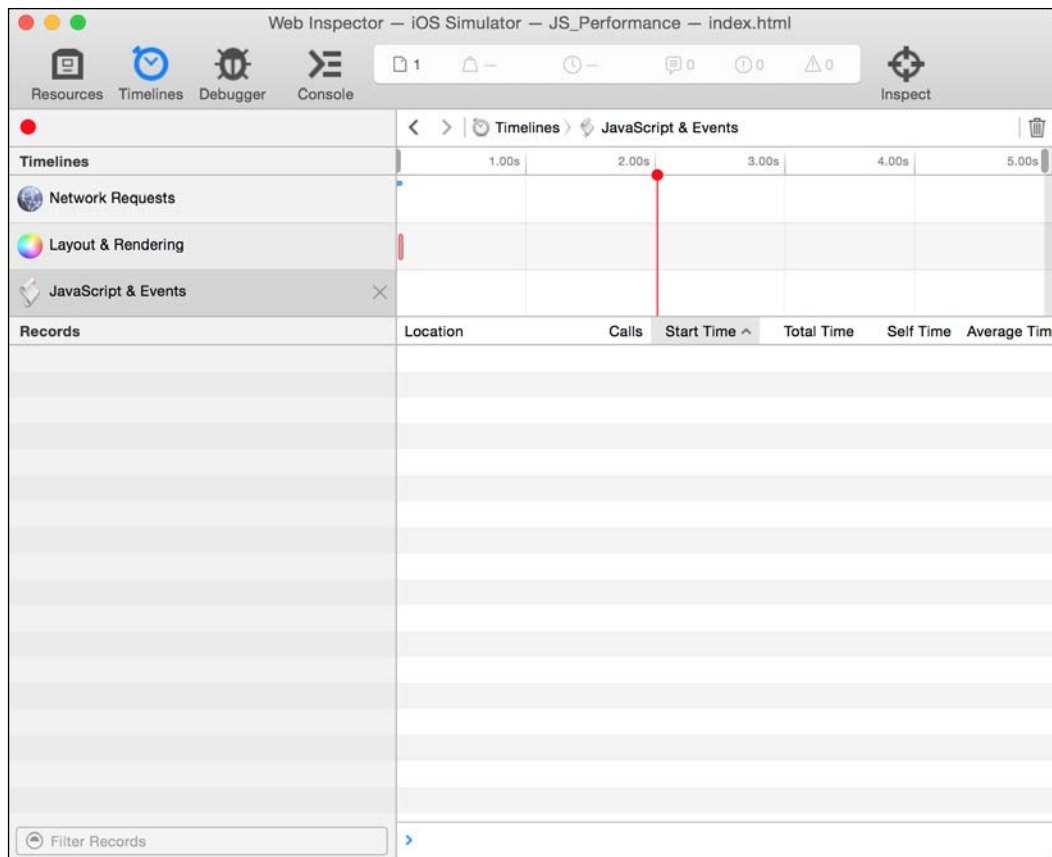
For a small app, this is minimal in terms of a difference in performance. But, as an application gets larger, the delay in these JavaScript processes gets longer and longer.

We can also debug the app using the debugging inspector in the Safari's **Web Inspector** tool. Click **Debugger** in the top menu panel in Safari's **Web Inspector**. We can add a break point to our embedded script by clicking a line number and then refreshing the page with *Command + R*. In the following screenshot, we can see the break occurring on page load, and we can see our scope variable displayed for reference in the right panel:



We can also check page load times using the timeline inspector. Click **Timelines** at the top of the **Web Inspector** and now we will see a timeline similar to the **Resources** tab found in Chrome's **Developer tools**. Let's refresh our page with *Command + R* on our keyboard; the timeline then processes the page.

Notice that after a few seconds, the timeline in the **Web Inspector** stops when the page fully loads, and all JavaScript processes stop. This is a nice feature when you're working with the Safari **Web Inspector** as opposed to Chrome's Developer tools.



Common ways to improve hybrid performance

With hybrid apps, we can use all the techniques for improving performance that we've learned in the past chapters: using a build system such as Grunt.js or Gulp.js with NPM, using JSLint to better optimize our code, writing code in an IDE to create better structure for our apps, and helping to check for any excess code or unused variables in our code.

We can use best performance practices such as using strings to apply an HTML page (like the `innerHTML` property) rather than creating objects for them and applying them to the page that way, and so on.

Sadly, the fact that hybrid apps do not perform as well as native apps still holds true. Now, don't let that dismay you as hybrid apps do have a lot of good features! Some of these are as follows:

- They are (typically) faster to build than using native code
- They are easier to customize
- They allow for rapid prototyping concepts for apps
- They are easier to hand off to other JavaScript developers rather than finding a native developer
- They are portable; they can be reused for another platform (with some modification) for Android devices, Windows Modern apps, Windows Phone apps, Chrome OS, and even Firefox OS
- They can interact with native code using helper libraries such as *Cordova*

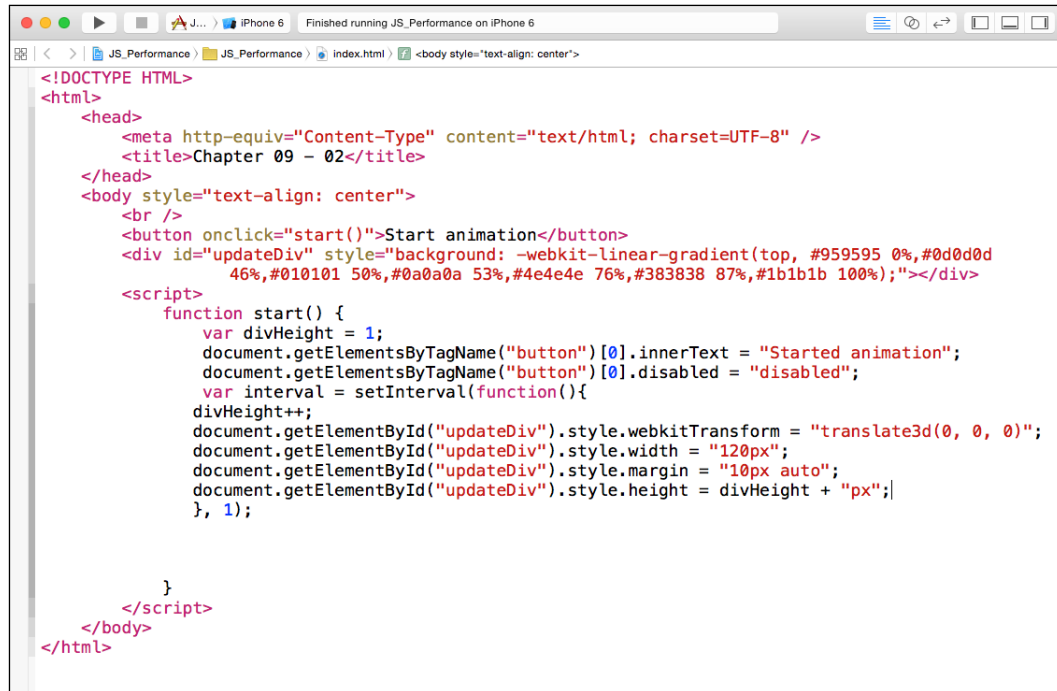
At some point, however, application performance will be limited to the hardware of the device, and it's recommended you move to native code. But, how do we know when to move? Well, this can be done using **Color Blended Layers**. The **Color Blended Layers** option applies an overlay that highlights slow-performing areas on the device display, for example, green for good performance and red for slow performance; the darker the color is, the more impactful will be the performance result.

Rerun your app using Xcode and, in the Mac OS toolbar for iOS Simulator, select **Debug** and then **Color Blended Layers**. Once we do that, we can see that our iOS Simulator shows a green overlay; this shows us how much memory iOS is using to process our rendered view, both native and non-native code, as shown here:



Currently, we can see a mostly green overlay with the exception of the status bar elements, which take up more render memory as they overlay the web view and have to be redrawn over that object repeatedly.

Let's make a copy of our project and call it `JS_Performance_CBL`, and let's update our `index.html` code with this code sample, as shown in the following screenshot:

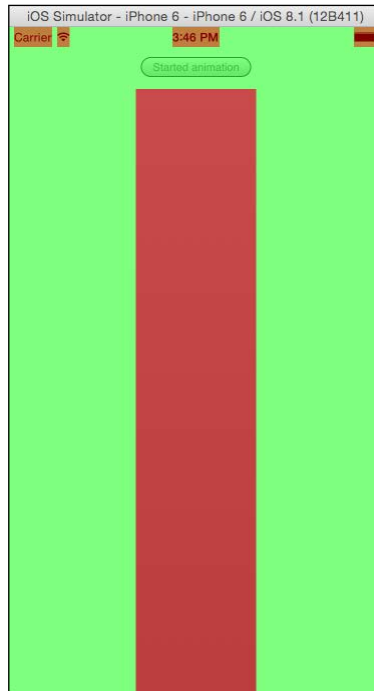


```
<!DOCTYPE HTML>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <title>Chapter 09 - 02</title>
  </head>
  <body style="text-align: center">
    <br />
    <button onclick="start()">Start animation</button>
    <div id="updateDiv" style="background: -webkit-linear-gradient(top, #959595 0%,#0d0d0d 46%,#010101 50%,#0a0a0a 53%,#4e4e4e 76%,#383838 87%,#1b1b1b 100%);"></div>
    <script>
      function start() {
        var divHeight = 1;
        document.getElementsByTagName("button")[0].innerText = "Started animation";
        document.getElementsByTagName("button")[0].disabled = "disabled";
        var interval = setInterval(function(){
          divHeight++;
          document.getElementById("updateDiv").style.webkitTransform = "translate3d(0, 0, 0)";
          document.getElementById("updateDiv").style.width = "120px";
          document.getElementById("updateDiv").style.margin = "10px auto";
          document.getElementById("updateDiv").style.height = divHeight + "px";
        }, 1);
      }
    </script>
  </body>
</html>
```

Here, we have a simple page with an empty div; we also have a button with an `onclick` function called `start`. Our `start` function will update the height continuously using the `setInterval` function, increasing the height every millisecond. Our empty div also has a background gradient assigned to it with an inline `style` tag.

CSS background gradients are typically a huge performance drain on mobile devices as they can potentially re-render themselves over and over as the DOM updates itself. Some other issues include listener events; some earlier or lower-end devices do not have enough RAM to apply an event listener to a page. Typically, it's a good practice to apply `onclick` attributes to HTML either inline or through JavaScript.

Going back to the gradient example, let's run this in **iOS Simulator** and enable **Color Blended Layers** after clicking our HTML button to trigger the JavaScript animation.



As expected, our div element that we've expanded now has a red overlay indicating that this is a confirmed performance issue, which is unavoidable. To correct this, we would need to remove the CSS gradient background, and it would show as green again. However, if we had to include a gradient in accordance with a design spec, a native version would be required.

When faced with UI issues such as these, it's important to understand tools beyond normal developer tools and Web Inspectors, and take advantage of the mobile platform tools that provide better analysis of our code. Now, before we wrap this chapter, let's take note of something specific for iOS web views.

The WKWebView framework

At the time of writing, Apple has announced the WebKit framework, a first-party iOS library intended to replace UIWebView with more advanced and better performing web views; this was done with the intent of replacing apps that rely on HTML5 and JavaScript with better performing apps as a whole.

The WebKit framework, also known in developer circles as **WKWebView**, is a newer web view that can be added to a project. WKWebView is also the base class name for this framework. This framework includes many features that native iOS developers can take advantage of. These include listening for function calls that can trigger native Objective-C or Swift code. For JavaScript developers like us, it includes a faster JavaScript runtime called *Nitro*, which has been included with Mobile Safari since iOS6.

Hybrid apps have always run worse than native code. But with the Nitro JavaScript runtime, HTML5 has equal footing with native apps in terms of performance, assuming that our view doesn't consume too much render memory as shown in our color blended layers example.

WKWebView does have limitations though; it can only be used for iOS8 or higher and it doesn't have built-in Storyboard or XIB support like UIWebView. So, using this framework may be an issue if you're new to iOS development. **Storyboards** are simply XML files coded in a specific way for iOS user interfaces to be rendered, while **XIB** files are the precursors to Storyboard. XIB files allow for only one view whereas Storyboards allow multiple views and can link between them too.

If you are working on an iOS app, I encourage you to reach out to your iOS developer lead and encourage the use of WKWebView in your projects.

For more information, check out Apple's documentation of WKWebView at their developer site at https://developer.apple.com/library/IOs/documentation/WebKit/Reference/WKWebView_Ref/index.html.

Summary

In this chapter, we learned the basics of creating a hybrid-application for iOS using HTML5 and JavaScript; we learned about connecting the Safari Web Inspector to our HTML page while running an application in iOS Simulator. We also looked at Color Blended Layers for iOS Simulator, and saw how to test for performance from our JavaScript code when it's applied to device-rendering performance issues.

Now we are down to the wire. As for all JavaScript web apps before they go live to a production site, we need to smoke-test our JavaScript and web app code and see if we need to perform any final improvements before final deployment. This is discussed in the next chapter.

10

Application Performance Testing

In this book, we've covered various ways of increasing our JavaScript's application performance at different stages of a project's life cycle. This includes activities ranging from choosing a proper editor at various stages in a project's lifespan, incorporating JavaScript linters to help proof our JavaScript before deployment to using build systems, and creating a deployment package or build separating final code from the developer-friendly code base.

The real secret in crafting high performing JavaScript is not the amount of JavaScript knowledge in our heads, but knowing the key "pain points" of the language itself; some of these pain points are the `for` loops, object creation, not incorporating strict operators, timers, and so on. Moreover, this category also includes incorporating these tools to better check our code before it is deployed.

Like all major web application projects, there is always some form of pre-flight check here, a final list of to-dos before a web application goes live. If we incorporate the tools covered in this book to this point, our JavaScript should be solid enough for deployment. But here, we will take it one step further.

In this chapter, we are going to take a look at **Jasmine**, a JavaScript testing framework that will allow us to test our code in ways we haven't realized yet. Unlike past linter tools such as, JSLint, these tests will rely on an application's property types, and also on a concept we have yet to cover: unit testing in JavaScript.

In short, we will be covering the following topics:

- What is unit testing in JavaScript?
- Unit testing with Jasmine

What is unit testing in JavaScript?

Unit testing, simply put, is an application framework or toolset designed to test JavaScript or the code of any other programming languages in a specific way that's unique to any application. Unit tests typically cover error checking that doesn't exist inside standard linters. They are designed to check for application-specific errors. In other programming languages, unit tests are typically designed to check a project's classes and models, and to ensure that applications are running efficiently and correctly.

Now JavaScript and unit-testing practices have never been associated well with one another, primarily due to the dynamic nature of JavaScript. Some factors that hamper their association include the many mistakes created unknowingly by developers, passing wrong values to variables that shouldn't have specific variable type, assigning a string when an application's object property requires a number, and so on.

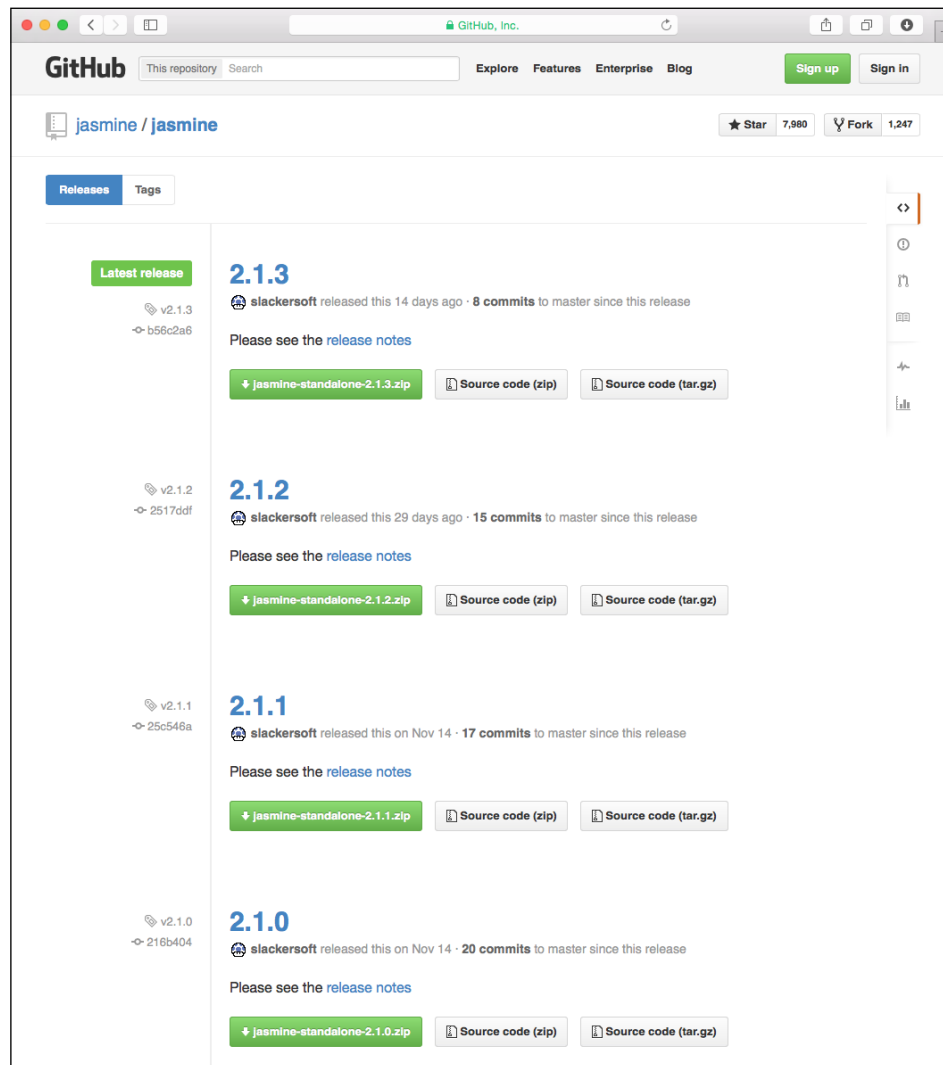
Moving forward, however, for client-side applications using JavaScript, whether they're on the web in a web browser or hosted inside a mobile app's web view, testing becomes more and more necessary. Now there are dozens of frameworks out there designed for JavaScript testing, but here, I will cover one in particular called Jasmine. Keep in mind that there are alternative testing frameworks such as Mocha or QUnit, but we will cover Jasmine as it doesn't require third-party frameworks to run.

Unit testing with Jasmine

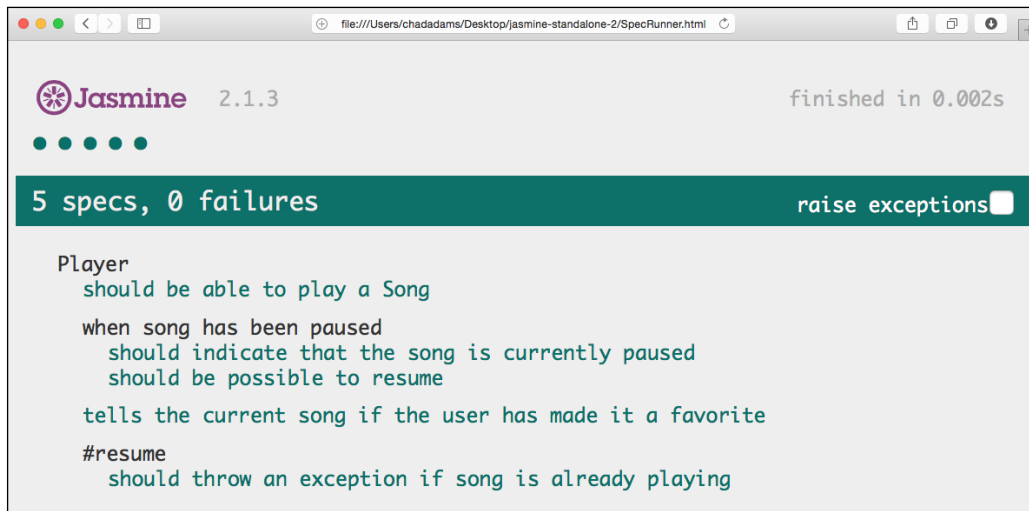
Jasmine is a JavaScript unit-testing framework; it allows us to write JavaScript without relying on external libraries such as jQuery. This is helpful for an application that requires a very tiny footprint in memory such as our JavaScript application in iOS, discussed in *Chapter 9, Optimizing JavaScript for iOS Hybrid Apps*. It also restricts the code to the code we've written, and there's no bug due to a framework in a current build of another vendor's library.

Installation and configuration

Jasmine can be installed in various ways; we can use node package manager or NPM similar to how we structure our Gulp.js build system in *Chapter 3, Understanding JavaScript Build Systems*. But, to get our feet wet with testing in general, we are going to download the standalone version of the framework. I will be using version 2.1.3, which is the latest stable release of the framework and can be found at <https://github.com/jasmine/jasmine/releases>. To download, click the green .zip file button shown here on the Jasmine framework's Github page:



Once we download the standalone version of Jasmine, we can check to see if it is working; the standalone version includes some sample JavaScript that's been set up with some unit tests. To run a set of unit tests in Jasmine, we will need to structure a SpecRunner page. A SpecRunner is a Jasmine-specific HTML page displaying the unit test results. If we open up the standalone versions `SpecRunner.html` file in our browser, we should see the example test results demonstrating all tests that have passed, as shown in the following screenshot:



Before setting up a test, we will need to test some code. I've created a bit of JavaScript that is object oriented and relies heavily on specific JavaScript types, such as numbers and Booleans, which are used throughout the application. The application is a very simple banking application that returns customer data to a simple HTML page, but it is structured enough to resemble a large application. We are going to use Jasmine to check for types, ensure that the data being passed in is valid, and the verify the application is outputting customer data as it should.

Reviewing the project code base

We will use the following code sample for the project. Take a moment and look through the code shown here. As always, all code samples for this book are available on Packt Publishing's website too.

```

1 // Gender Enumeration
2 var Gender;
3 (function (Gender) {
4     Gender[Gender["Male"] = 0] = "Male";
5     Gender[Gender["Female"] = 1] = "Female";
6     Gender[Gender["Alien"] = 2] = "Alien";
7 })(Gender || (Gender = {}));
8
9 // @class BankDB - Client-side database class object, with common query functions.
10 var BankDB = (function () {
11     function BankDB(_customer) {
12         this.customerID = _customer.customerID;
13         this.customerBalance = _customer.customerBalance;
14         this.customerName = _customer.customerName;
15         this.customerCity = _customer.customerCity;
16         this.customerGender = _customer.customerGender;
17         this.customerMarried = _customer.customerMarried;
18     }
19
20     BankDB.prototype.requestCustomerCityName = function () {
21         return "City: " + this.customerCity;
22     };
23
24     BankDB.prototype.requestBankBalance = function () {
25         var stringBalance = "Balance: $" + this.customerBalance;
26         return stringBalance;
27     };
28
29     BankDB.prototype.requestCustomerGreeting = function () {
30         var stringBalance;
31         switch (this.customerGender) {
32             case 0:
33                 stringBalance = "Hello Mr. " + this.customerName[1] + ".";
34                 break;
35             case 1:
36                 if (this.customerMarried) {
37                     stringBalance = "Hello Mrs. " + this.customerName[1] + ".";
38                 }
39                 else {
40                     stringBalance = "Hello Miss. " + this.customerName[1] + ".";
41                 }
42                 break;
43             case 2:
44                 stringBalance = "Live long and prosper " + this.customerName[1] + ".";
45                 break;
46             default:
47                 stringBalance = "Hello " + this.customerName[1] + ".";
48                 break;
49         }
50         return stringBalance;
51     };
52     return BankDB;
53 })();
54
55 // New Customer data value
56 var newCustomer = {
57     customerID: 54323421,
58     customerName: "Leonard Adams",
59     customerBalance: "40000",
60     customerCity: "Raymore",
61     customerGender: 0 /* Male */,
62     customerMarried: 'false'
63 };
64
65 // New request to the client-side database.
66 var request = new BankDB(newCustomer);
67 document.body.style.fontFamily = "Segoe UI", Helvetica, Arial, sans-serif";
68 document.body.style.fontSize = "2em";
69 document.body.style.textAlign = "center";
70 document.body.innerHTML += "<br />" + request.requestCustomerGreeting();
71 document.body.innerHTML += "<br />" + request.requestCustomerCityName();
72 document.body.innerHTML += "<br /><strong>" + request.requestBankBalance() + "</strong>";
73

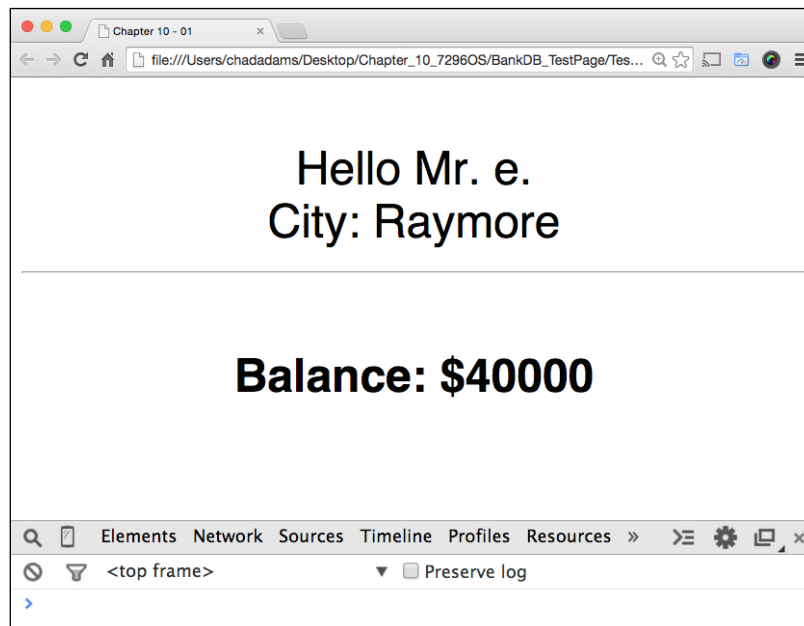
```


We have quite a bit of code here to test, but there's no need to worry about that! Let's review this slowly before we start using Jasmine. On lines 1 through 7, we have a JavaScript enumeration for a gender type allowing us to predefine values for a customer type. In this example, the values are either `Male`, `Female`, or `Alien`. Starting on line number 10 is our `BankDB` object (also considered a JavaScript class); now this isn't really a database, but it could very well be connected to one in a real application.

The `BankDB` function is an instance-based object, meaning it requires a specific type of parameter in order for it to function, that we can find on line 56 called `newCustomer`. This JavaScript object contains a JavaScript object notation, which assigns values to a new customer entry. Think of this as a bit of JSON being returned by a clerk while using the system.

Finally, on lines 66 through 72, we create the request with that user's data, and then append the data to the embedded web page's `document.body` statement with a bit of light styling and formatting.

Before we start writing our tests, let's look at this in a self-contained page. I'll add this to an empty HTML page just before the closing `body` tag. Let's open the page and look at the results, which resemble the following screenshot:



As we can see, our application is displaying all the correct information except the customer's name, which is showing as Mr. e, rather than Mr. Leonard Adams as indicated back on line 58 of our `10_01.js` file. Also, notice that, in our Chrome **Developer tools** option, we are not receiving any errors, and not really seeing much of a performance lag either. Nevertheless, we do know by the output of the customer's name that something is wrong. To correct this, we will unit-test our application.

Reviewing an application's spec for writing tests

When writing unit tests, there need to be well-defined instructions for writing the tests; in the case of the code sample shown in the previous screenshot, we want to ensure that our tests follow a few rules and, to help us write these tests, we'll use the rules listed in the following table with our code.

Consider the following list as an application specification, or documentation based on which the application should be built. Let's look at the table and see what our code should be doing with the data being used:

Test Number	Test Description
Test #1	New Customer data test: Customer's ID should be a number.
Test #2	New Customer data test: Customer's name should be in an array object, (ex ['FirstName', 'LastName']).
Test #3	New Customer data test: Customer's bank balance should be a number.
Test #4	New Customer data test: Customer's city name should be a string.
Test #5	New Customer data test: Customer's gender should be a number.
Test #6	New Customer data test: Customer's marriage status is a boolean.

According to this list, we need our data values to pass these six tests in order to ensure that the JavaScript application is working properly. To do this, we will write a **spec** using Jasmine. In the Jasmine framework, a spec file is simply a JavaScript file with the JavaScript to be tested loaded into an HTML page that contains both the Jasmine testing framework and the file to be tested. Here, we can see what that combined page looks like; in Jasmine-based testing, it is typically called a SpecRunner page:

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

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

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

    <!-- include source files here... -->
    <script src="src/Chapter_10_01.js"></script>

    <!-- include spec files here... -->
    <script src="spec/Chapter_10_01Spec.js"></script>

  </head>

  <body>
  </body>
</html>
```

Here, we can see the SpecRunner.html page and notice that we have the Jasmine frameworks loaded first in the head tag, followed by our test script shown earlier in the chapter called Chapter_10_01.js, which is then followed by our spec file named as Chapter_10_01_Spec.js for consistency.

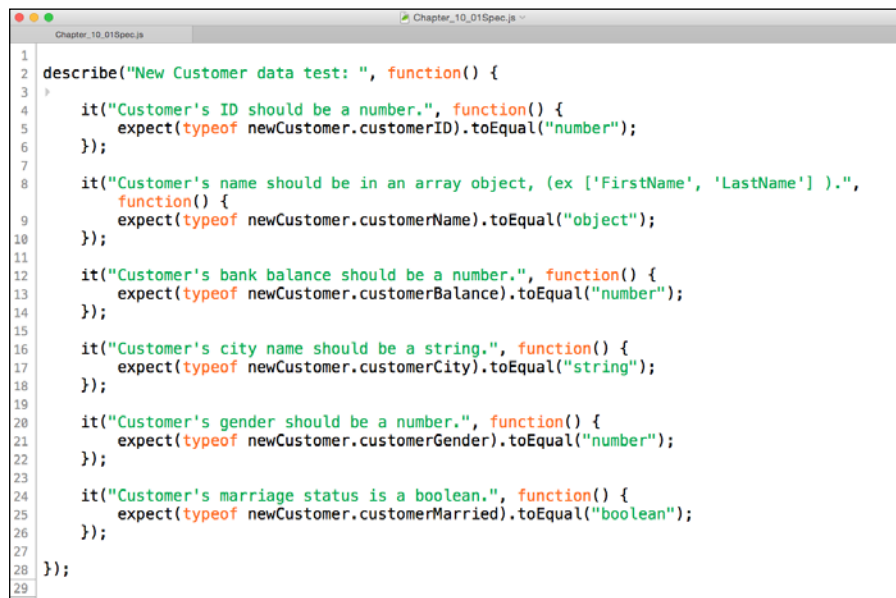
Note that if, we open our Chrome **Developer tools** in our SpecRunner.html page, we can see a few errors coming from our 10_01.js file where we append the document.body statement with our customer data. JavaScript that uses the DOM may cause issues with Jasmine and any other JavaScript testing frameworks as well, so be sure to use application-specific code to test rather than a user interface code.

Writing tests using Jasmine

In Jasmine, there are three keywords specific to the testing framework that we need to know. The first is `describe`; `describe` is like a class in testing. It groups our tests in one container to be referenced later. In the previous list from our application spec, we can use `New Customer data test` as our `describe` value.

The second keyword is `it`; `it` is a Jasmine function that takes two parameters, a string that we use as our test description. For example, one `it` test could contain a description such as `Customer's ID should be a number`. This tells the user reviewing the test what exactly we are testing for. The other parameter is a function where we can inject code or set up code if needed. Remember that all of this is being run in the same page, so if we would like to change any variables, or prototypes for a test, we can do that within this function before we run our test. Keep in mind that, while writing the test, we don't need to modify our code in order to test properly; this is done only in case we don't have a code sample for review.

The last keyword to remember is `expect`; `expect` is a function specific to Jasmine that takes a value and compares it with some other value. In Jasmine, this is done using the `toEqual` function that is a part of the `expect` function. Think of each test as readable like this: We expect the `typeof newCustomer.customerID` to equal a number. Now this is pretty simple if we think about it, but what does that look like in a spec file? Well, if we look at the following screenshot, we can see our `Chapter_10_01Spec.js` file with each of the tests written ready for Jasmine:

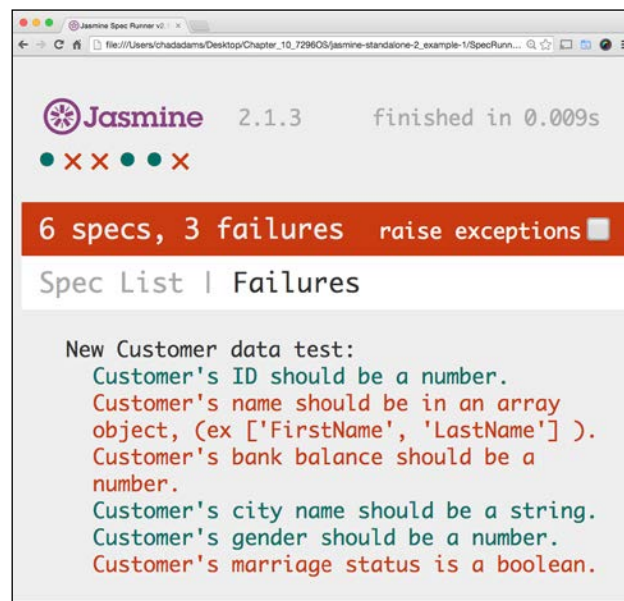


```
1 describe("New Customer data test: ", function() {
2
3   it("Customer's ID should be a number.", function() {
4     expect(typeof newCustomer.customerID).toEqual("number");
5   });
6
7   it("Customer's name should be in an array object, (ex ['FirstName', 'LastName'] ).",
8     function() {
9     expect(typeof newCustomer.customerName).toEqual("object");
10    });
11
12   it("Customer's bank balance should be a number.", function() {
13     expect(typeof newCustomer.customerBalance).toEqual("number");
14   });
15
16   it("Customer's city name should be a string.", function() {
17     expect(typeof newCustomer.customerCity).toEqual("string");
18   });
19
20   it("Customer's gender should be a number.", function() {
21     expect(typeof newCustomer.customerGender).toEqual("number");
22   });
23
24   it("Customer's marriage status is a boolean.", function() {
25     expect(typeof newCustomer.customerMarried).toEqual("boolean");
26   });
27 });
28
29
```

Here, we can see how our tests are written; on line 2, we have our `describe` keyword that wraps our tests in a single container should we have a larger test file. All our tests from our documentation spec can be found with each `it` keyword; test number 1 is on line 4 and, on line 5, we have the first test's `expect` keyword checking the `newCustomers.CustomerID` type, where we expect a number.

Note that the type being compared is using a string rather than number, as you would expect in a console. This is because `typeof`, the JavaScript keyword for returning the type of a variable or property, returns the type name using a string; so, in order to match it, we use strings with the type name here as well.

We can see on subsequent lines that we've added the remaining tests using the same style of comparison for each of the other tests. With that done, let's open the `SpecRunner.html` page; we can see how our tests did in the **Spec List** view in the following screenshot:



Yikes! Three errors here, which is not good at all. Here, we were expecting a single error with the name of the customer not being displayed properly. But, our unit test has discovered that our application spec isn't being followed as it was written. In the Jasmine framework, this page layout is pretty common; on initial load, you will see a full error list. If you wish to see the list of all the tests that passed and failed, we can click **Spec List** at the top, and we will see the full list as shown in the preceding screenshot.

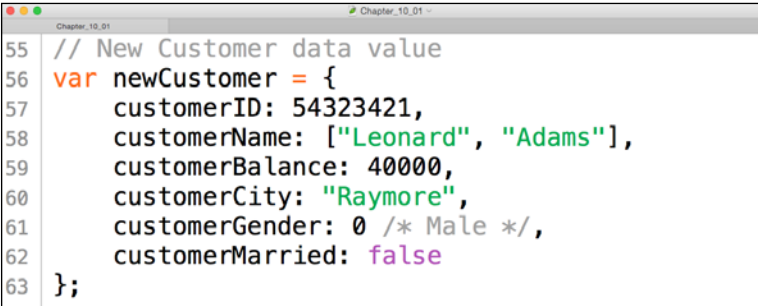
Tests that have failed here show up in red on your browser, and those that have passed show up in green. You may also see green circles and red Xs indicating how many tests passed and failed in both the **Failures** view and the **Spec List** view.

Fixing our code

With our test code working now, we can modify it to ensure this works properly. For this, we will need to update the `10_01.js` file and the `newCustomer` data, which is on lines 56 through 63 in the `10_01.js` file. Let's review what went wrong with our sample customer data:

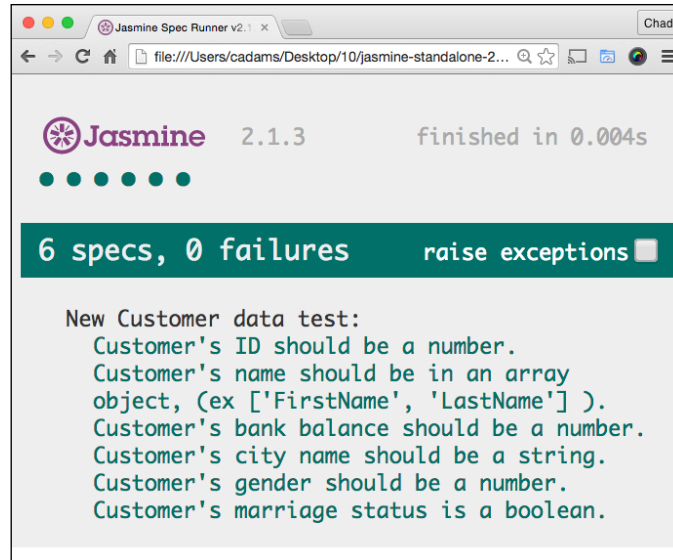
- The first test that failed was 2, which required the customer's name to be created as an object array, with the first name as an array item followed by the last name as the second item in the object array
- The second that failed was test 3, which required the `customerBalance` to be a type of number
- The third error was test 6, which required the customer's marriage status to be a boolean and not a string

Let's update our `newCustomer` data; you can see that I've done that in the following screenshot:

A screenshot of a code editor window titled 'Chapter_10_01'. The editor shows a JavaScript code snippet for a new customer. The code is as follows:

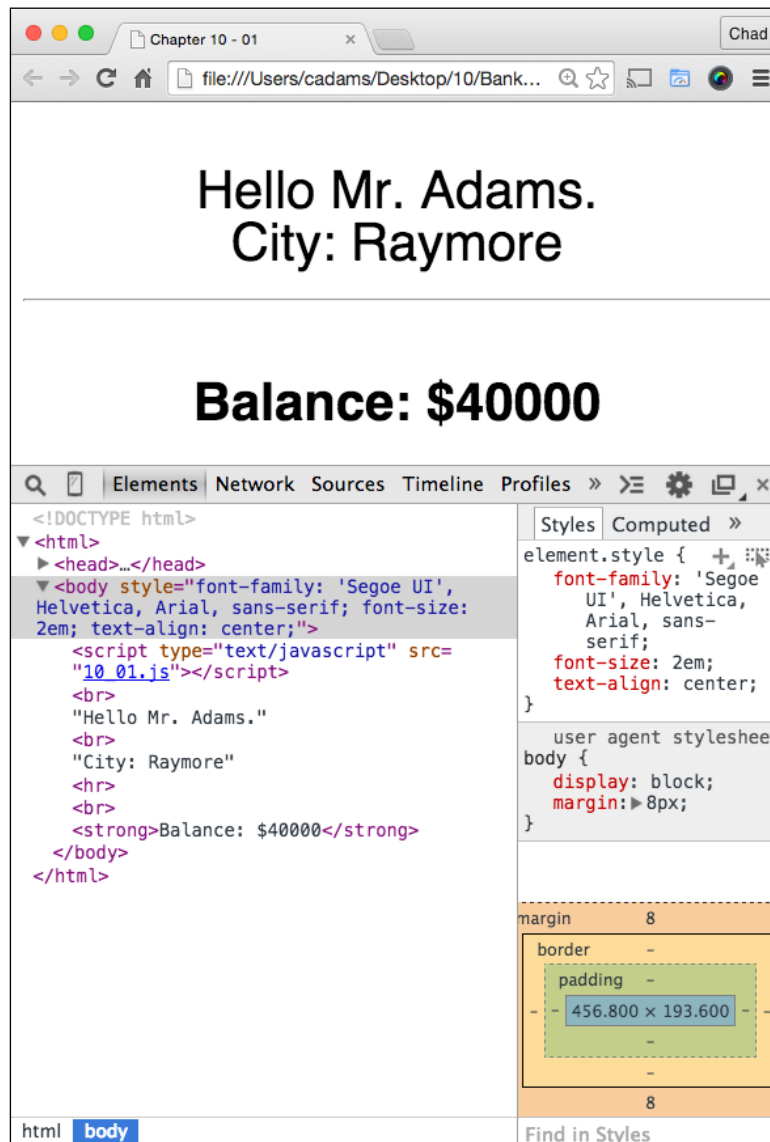
```
55 // New Customer data value
56 var newCustomer = {
57   customerID: 54323421,
58   customerName: ["Leonard", "Adams"],
59   customerBalance: 40000,
60   customerCity: "Raymore",
61   customerGender: 0 /* Male */,
62   customerMarried: false
63 };
```

Once we've updated the `newCustomer` information in our `10_01.js` file, we should be able to rerun Jasmine and retest our code sample. If all tests pass, we will see the default **Spec List** showing all results in green; let's reopen our page as shown in the following screenshot and see whether our tests pass:



Nice, all six specs have passed! Great work! By ensuring that all our application's data is using the correct type, we can also ensure that our JavaScript application is not only performing well but also performing with a high degree of accuracy, as it was intended to be used.

When applications deviate from the developers, design, they can cause performance issues and affect the overall stability of the application. In Jasmine, we can see the completion time of the test; note that the performance on the final test is much faster than the one with errors. In the following screenshot, we have our final application page with no errors, as shown by **Developer tools** option in Chrome:



One final fact to note here is the different approaches that can be used by JavaScript developers. One is the **Test Driven Development (TDD)** approach, where we write our tests before writing our application code. Another way in which many JavaScript developers test applications is called **Behavior Driven Development (BDD)** approach. This works by writing app code first and then interacting with an app, which includes opening a popup and confirming that the code worked as intended.

Both of these are valid methods to build applications, but for JavaScript applications, which use a bit of data that must be accurate, TDD is the way to go!

Summary

In this chapter, we covered the basics of unit testing JavaScript applications. We introduced Jasmine, a behavior-driven unit-testing framework for JavaScript. Together we created a real-world application that had no technical errors but was causing application issues.

We reviewed how to read and write an application spec and how to write tests in Jasmine using the applications spec. We then ran our test against our code and quickly updated our customer data to reflect the spec, allowing our unit test to pass. Lastly, we learned that unit-testing our code improves our JavaScript performance, and also minimizes risk to our application.

Index

A

Apple

URL, for documentation 60

Apple's iOS Developer Center documentation

URL 152

array performance 108

array searches

optimizing 109-111

Asynchronous JavaScript and XML (AJAX) 114

Audits panel, Chrome's Developer tools

about 78

interacting with 79

suggestions, obtaining for
JavaScript quality 80

B

Behavior Driven Development (BDD) 182

build system

about 35

code, compiling by example 36

distribution, creating 56, 57

error checking 37, 38

example file, testing 54-56

optimization, adding beyond coding
standards 38

setting up 47

C

Canary

URL, for downloading 63

Chrome

URL 64

Chrome's Developer tools

about 62, 63

Audits panel 78, 79

Console panel 81, 82

Elements panel 66

Network panel 67-69

overview 64-66

Profile panel 76

Resources panel 77

Sources panel 70

Timeline panel 74, 75

cloud-based editors

about 12

Cloud9 editor 12, 13

Codenvy editor 14

code performance, JavaScript

checking 17

comparison operator

about 84

example 84, 85

compiler 35

console, JSLint 32, 33

Console panel, Chrome's Developer tools

about 81

URL 82

console.time API 18-21

console.time() method 21

constructor 95

constructor functions

versus prototypes 107, 108

createElement function

new objects, creating with 115

using 120

working with 115-119

CSS3

used, for animating elements 122, 123

D

debugger

- about 70
- testing 70-73
- using 73, 74

Document Object Model (DOM) 113, 114

E

effective editor

- selecting 3

elements, animating

- about 120-122
- CSS3 used 122, 123
- unfair performance advantage 124-126

Elements panel, Chrome's Developer tools 66

F

Firefox Developer tools 60, 61

G

graphics processing unit (GPU) 126

Grunt.js 47

Grunt Task Runner 48

Gulp

- about 48
- installing 49, 50
- JSLint, integrating into 53, 54
- URL, for plugins 49

gulpfile

- creating 51

Gulp.js

- about 47
- used, for creating build system 38

Gulp project

- running 52

I

instance functions 98

instances

- about 101
- creating, with new keyword 101-105

Integrated Development Environments (IDEs)

about 4

JetBrain's WebStorm IDE 6, 7

Microsoft Visual Studio IDE 4, 5

Internet Explorer developer tools

about 61, 62

URL 62

Internet Service Provider (ISP) 1

iOS development 149

iOS hybrid development

about 150-152

Safari Web Inspector, using for JavaScript performance 158-160

simple iOS hybrid app, setting up 153-158

UIWebView, versus Mobile Safari 161, 162

ways, for improving performance 163-166

WKWebView framework 166, 167

J

Jasmine

about 169, 170

configuring 171, 172

installing 171, 172

unit testing 170

URL, for releases 171

used, for writing tests 177, 178

JavaScript

about 3

code performance, checking 17

Safari Web Inspector, using for performance 158-160

unit testing 170

JetBrain's WebStorm IDE 6, 7

jQuery

Node Package Manager (NPM), installing with 45-47

JSLint

about 22

console 32, 33

errors, reviewing 26

integrating, into Gulp 53, 54

messy white space, configuring 27-29

URL 23

use strict statement 30, 31

using 24-26

L

lightweight editors

- about 10
- Notepad++ editor 11
- Sublime Text editor 10, 11

local server

- worker, testing with 140-142

loops

- about 86
- performance, affecting 86-88
- reverse loop performance myth 89-92

M

Microsoft Visual Studio IDE 4, 5

Microsoft WebMatrix editor 9

mid-range editors

- about 8
- Microsoft WebMatrix editor 9
- Panic's Coda editor 8, 9

Mobile Safari

- versus UIWebView 161, 162

Model View Controller (MVC) 83

Mozilla's Developer Network

- URL 61

N

Network panel, Chrome's

- Developer tools 67-69

new keyword

- instances, creating with 101-105

new objects

- creating, with createElement function 115

Node.js

- about 39-41
- installation, testing 41, 42
- URL 40

Node Package Manager (NPM)

- about 43
- installation, checking in Terminal 44
- installation, testing 43
- jQuery, installing with 45-47
- URL 43
- using 44, 45

Notepad++ editor 11

O

operators

- about 84
- comparison operator 84, 85

P

paint events

- about 126
- checking for 126, 127
- testing 128, 129

Panic's Coda editor 8, 9

pesky mouse scrolling events 129-132

Profile panel, Chrome's Developer tools 76

promises

- about 96, 142-144
- reference link 96
- true asynchronous promise, testing 144-146

prototypes

- about 98
- in terms, of memory 106
- versus constructor functions 107, 108

R

Resources panel, Chrome's Developer

- tools 77

reverse loop performance myth 89-92

S

Safari Web Inspector

- about 60
- using, for JavaScript performance 158-160

scope 102

simple iOS hybrid app

- setting up 153-158

Sources panel, Chrome's Developer tools

- about 70
- debugger 70

spec 176

storyboards 167

Sublime Text editor 10, 11

Swift 153

T

Test Driven Development (TDD) 182

tests

writing, Jasmine used 177, 178

this keyword 102

Timeline panel, Chrome's Developer tools

about 74

Loading event 75

Painting event 76

Rendering event 76

Scripting event 76

using 74

timers

about 92

performance, affecting 93, 94

single threading 94-96

U

Uglify

URL 56

UIWebView

about 153

versus Mobile Safari 161, 162

unit testing, in Jasmine

about 170

application's spec for writing tests,
reviewing 175, 176

code, fixing 179-182

project code base, reviewing 173-175

tests, writing 177, 178

unit testing, in JavaScript 170

use strict statement 30, 31

W

Web Inspectors

about 59

Chrome's Developer tools 62, 63

Firefox Developer tools 60, 61

Internet Explorer developer tools 61, 62

Safari Web Inspector 60

web workers 134-139

WKWebView framework

about 166

URL 167

worker

about 134

testing, with local server 140-142

X

Xcode

about 150

installing 150-152

XIB files 167



Thank you for buying **Mastering JavaScript High Performance**

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.



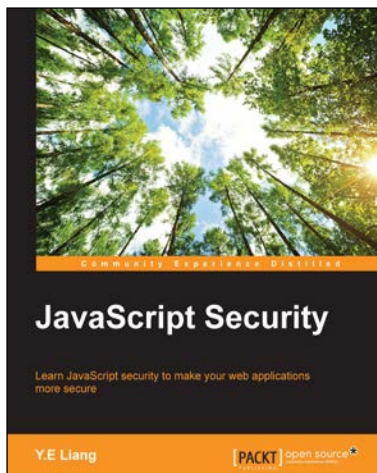
Learning Object-Oriented JavaScript

ISBN: 978-1-78355-433-1

Duration: 02:47 hours

Acquire advanced JavaScript skills and create complex and reusable applications

1. Discover the important concepts of object-oriented programming (OOP) and make your life easier, more enjoyable, and more focused on what you love doing—creating.
2. Develop reusable code while creating three different clocks, a classic clock, a text clock, and an alarm clock.
3. Utilize the advantages of using constructors, methods, and properties to become an expert.



JavaScript Security

ISBN: 978-1-78398-800-6

Paperback: 112 pages

Learn JavaScript security to make your web applications more secure

1. Understand the JavaScript security issues that are a result of both the frontend and the backend of a web app.
2. Learn to implement Security techniques to avoid cross site forgery and various JavaScript vulnerabilities.
3. Secure your JavaScript environment from the ground up, with step-by-step instructions covering all major ways to tackle Security issues.

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



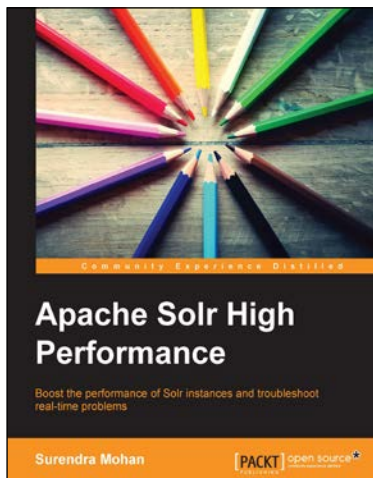
JavaScript Mobile Application Development

ISBN: 978-1-78355-417-1

Paperback: 332 pages

Create neat cross-platform mobile apps using Apache Cordova and jQuery Mobile

1. Configure your Android, iOS, and Window Phone 8 development environments.
2. Extend the power of Apache Cordova by creating your own Apache Cordova cross-platform mobile plugins.
3. Enhance the quality and the robustness of your Apache Cordova mobile application by unit testing its logic using Jasmine.



Apache Solr High Performance

ISBN: 978-1-78216-482-1

Paperback: 124 pages

Boost the performance of Solr instances and troubleshoot real-time problems

1. Achieve high scores by boosting query time and index time, implementing boost queries and functions using the Dismax query parser and formulae.
2. Set up and use SolrCloud for distributed indexing and searching, and implement distributed search using Shards.
3. Use GeoSpatial search, handling homophones, and ignoring listed words from being indexed and searched.

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