

Gulp

Community Experience Distilled

Getting Started with Gulp

Create powerful automations with gulp to improve the efficiency of your web project workflow

Foreword by Eric Schottstall, Creator of Gulp

Travis Maynard

[PACKT] open source*
PUBLISHING
community experience distilled

Getting Started with Gulp

Create powerful automations with gulp to improve the efficiency of your web project workflow

Travis Maynard



BIRMINGHAM - MUMBAI

Getting Started with Gulp

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: January 2015

Production reference: 1200115

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

ISBN 978-1-78439-576-6

www.packtpub.com

Gulp image sourced from <http://gulpjs.com/>

Credits

Author

Travis Maynard

Project Coordinator

Danuta Jones

Reviewers

Raymond, Xie Liming
Dmitri Moore
Ranganadh Paramkusam
Konstantin Tarkus
Juris Vecvanags

Proofreaders

Simran Bhogal
Samuel Redman Birch
Ameesha Green
Paul Hindle

Commissioning Editor

Ashwin Nair

Indexers

Monica Ajmera Mehta
Rekha Nair

Acquisition Editor

Sonali Vernekar

Production Coordinator

Shantanu N. Zagade

Content Development Editor

Arwa Manaswala

Cover Work

Shantanu N. Zagade

Technical Editor

Bharat Patil

Copy Editors

Gladson Monteiro
Merilyn Pereira

Foreword

When I sat down to write about gulp, I had absolutely no idea how big it would be. Here we are, almost a year later, and the project has grown into one of the most popular libraries in JavaScript. It's been crazy to hear from our users how it has been benefiting them on a day-to-day basis: it improves developer workflow, saves a massive amount of time, and automates tedious tasks to restore sanity. I'm proud that the project has also introduced a new way of modeling data transformations (Streams!) to a new group of people and changed the way we think about build systems. Travis Maynard is one of our earliest adopters. When gulp had only 30 stars and nobody really cared about it, Travis was there writing amazing articles to guide people into a state of build system bliss. I can think of no better person to write this book. Go forth and build!

Eric Schöffstall
Creator of Gulp

About the Author

Travis Maynard is a frontend web developer who focuses on minimalism and simplicity. He is currently creating software at The Resumator to help make the hiring process a more successful and enjoyable experience.

Prior to his work at The Resumator, he created efficient web frameworks and user experiences at West Virginia University to serve their education and marketing efforts.

In his spare time, he actively contributes to the open source software community and occasionally writes about the web on his website (<http://travismaynard.com>).

I would like to thank the gulp team for creating and supporting such a great tool. It has changed the lives of many developers and made our workflows far more efficient and enjoyable. More specifically, I would like to thank the team members Eric Schöffstall and Blaine Bublitz for their help with code reviews and mentoring me throughout my initial writings on gulp.

A big thanks to Sonali Vernekar, Arwa Manasawala, Neeshma Ramakrishnan, and Bharat Patil along with everyone at Packt Publishing for giving me the opportunity to write this book and mentoring me throughout the writing process.

I would also like to thank Alaina Maxwell for supporting me throughout the many late nights of writing, and for reviewing and reading through page upon page of content to offer suggestions that undoubtedly make this book far more enjoyable to read.

Last, but most certainly not least, I would like to thank you, the reader, for choosing this book as your introductory guide to gulp. I hope that it serves you well, and what you learn will benefit you for many years to come.

About the Reviewers

Raymond, Xie Liming is a software R&D expert with over 16 years of experience working in multiple IT domains, including networking, multimedia IP communication, insurance, telecom, and mobile apps/games.

Raymond holds a Master of Science degree from Fudan University. He is also a PMI-certified Project Management Professional.

He worked as a senior manager in Softfoundry, Singapore; eBaoTech; and Ericsson's Shanghai R&D center, leading the R&D team while working on enterprise and carrier class software. In December 2013, Raymond founded his own company, RJFUN Mobile Technologies, that makes mobile apps/games and also produces reusable components for mobile apps/games.

Raymond has plenty of experience in R&D management. He is also a software expert with hands-on architecting and developing skills. He is very active in the Github and Cordova/PhoneGap communities with the nickname floatinghotpot.

Raymond now lives with his wife, Jenny, in Shanghai, China.

Dmitri Moore is a full stack software architect , specifically dealing with Javascript, and a hands-on developer with a primary focus on solutions based on AngularJS/Node.js. Working as an independent consultant, Dmitri has assisted many clients with building their IT infrastructure and implementing mission-critical apps.

In his spare time, apart from contemplating "2 spaces versus 4 spaces," Dmitri contributes to open source projects and shares his software-related thoughts on his web blog (<http://demisx.github.io>).

Ranganadh Paramkusam holds a degree in Computer Science and Engineering. He began his career by developing cross-platform applications for iOS, Android, and BlackBerry using PhoneGap, Sencha, and AngularJS respectively. He also developed more than 30 mobile applications.

Afterwards, he started working with native code, such as iOS and Java, to create PhoneGap plugins to introduce native UI/UX in hybrid mobile applications.

Ranganadh developed plugins using Google's Native Client (NaCl), and more specifically the Portable Native Client (PNaCl), to create web applications in a way that their performance would be similar to that of desktop applications. He also created browser extensions for Google Chrome and Firefox using Google APIs.

His works include creating a web-based image editor and a text editor (a replica of Adobe's Brackets application). He created a web-based image editor using the HTML5 Canvas element to apply enhance, filters, resize, and various other effects, and a chat application using Node.JS and MongoDB.

Ranganadh has a certification as an Oracle Certified Associate (OCA), which he got in 2010, and Python from MIT in 2013.

He was awarded the Techno Geek for the year 2012-13 and Emerging Performer of the Year (2013-14) for his work.

His whole work aims at linking JavaScript to low- and medium-level languages, and he came to JavaScript after developing for C++, Python, Objective-C, and Java.

In his leisure time, he reviews Packt Publishing books. His last book was *JavaScript Native Mobile Application Development*.

He is currently working as a senior programmer in the Center of Excellence (COE) department of Hidden Brains Infotech Pvt. Ltd., India.

I would like to thank my family and friends for their support while I was working on this book.

Konstantin Tarkus has been designing and building web applications for more than a decade. He has worked with small startups and global brands. He currently runs a small software consultancy company, Kriasoft, and actively contributes to the open source community. You can reach out to him at <https://www.codementor.io/koistya>

Juris Vecvanags started his career in the IT field in the early 90s. During this time, he had a chance to work with a broad range of technologies and share his knowledge with Fortune 500 companies as well as private and government customers.

Before moving to Silicon Valley, he had a well-established web design company in Europe. He is currently working as a solutions architect at Sencha Inc., helping customers write better apps for both desktops and emerging mobile platforms. He contributes to the ExtJs framework as well as writing custom components and features.

When it comes to web technologies, this invaluable experience serves as the basis to be a trusted advisor and competent reviewer.

Away from office, you can see him speaking at meetups across the San Francisco Bay area, Chicago, and New York. Among the topics he discusses are Node.js, ExtJs, and Sencha Touch.

He is passionate about bleeding edge technologies and everything related to JavaScript.

I would like to thank my family for their constant support while I was working on this book.

www.PacktPub.com

Support files, eBooks, discount offers, and more

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

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

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



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

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

Why subscribe?

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

Free access for Packt account holders

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

Table of Contents

Preface	1
Chapter 1: Introducing Gulp	5
What is gulp?	6
What is node.js?	7
Why use gulp?	7
Project automation	7
Streams	9
Code over config	9
Summary	10
Chapter 2: Getting Started	11
Getting comfortable with the command line	11
Command reference	14
Listing files and folders (ls)	14
Changing directory/folder (cd)	15
Making a directory/folder (mkdir)	16
Creating a file on Mac/Linux (touch)	17
Creating a file on Windows (ni)	17
Administrator permissions (sudo)	18
Creating your project structure	18
Adding content to the project	24
Preparing our HTML file	24
Preparing our CSS	25
Preparing our JavaScript	26
Adding images	27
Installing node.js and npm	27
Downloading and installing node.js	27
Verifying the installation	28
Creating a package.json file	29

Table of Contents

Installing gulp	30
Locating gulp	30
Installing gulp locally	31
Installing gulp globally	33
Anatomy of a gulpfile	33
The task() method	33
The src() method	34
The watch() method	34
The dest() method	34
The pipe() method	34
The parallel() and series() methods	34
Including modules/plugins	35
Writing a task	36
Reflection	37
Summary	38
Chapter 3: Performing Tasks with Gulp	39
Using gulp plugins	39
The styles task	40
Installing gulp plugins	40
Including gulp plugins	42
Writing the styles task	42
Other preprocessors	44
Reviewing the styles task	45
The scripts task	45
Installing gulp plugins	45
Including gulp plugins	47
Writing the scripts task	47
Reviewing the scripts task	50
The images task	50
Installing gulp plugins	50
Including gulp plugins	52
Writing the images task	52
Reviewing the images task	54
The watch task	54
Writing the watch task	54
Reviewing the watch task	57
The default task	57
Writing the default task	57
Completed gulpfile	58

Table of Contents

Running tasks	59
Running the default task	59
Running a single task	59
Stopping a watch task	60
Summary	60
Chapter 4: Using Node.js Modules for Advanced Tasks	61
Why use plain node.js modules?	61
Static server	62
Installing modules	62
Including modules	63
Writing a server task	64
BrowserSync	66
Installing BrowserSync	66
Including BrowserSync	67
Writing the BrowserSync task	68
Browserify	71
Installing modules	71
Including modules	72
Writing the Browserify task	73
Summary	75
Chapter 5: Resolving Issues	77
Handling errors	77
Installing gulp-plumber	78
Including gulp-plumber	78
Installing beeper	80
Including beeper	81
Writing an error helper function	82
Source ordering	84
Project cleanup	85
Installing the del module	85
Including the del module	85
Writing a clean task	86
External configuration	87
Task dependencies	88
Source maps	90
Installing a source maps plugin	90
Including a source maps plugin	90
Adding source maps to the PipeChain task	92
Summary	93

Table of Contents

Appendix: Key Features and Additional Resources	95
Chapter 1 – Introducing Gulp	95
Chapter 2 – Getting Started	95
Chapter 3 – Performing Tasks with Gulp	96
Chapter 4 – Using Node.js Modules for Advanced Tasks	96
Chapter 5 – Resolving Issues	97
References	97
Stream-handbook	97
NodeSchool	97
Gulp recipes	98
Index	99

Preface

I wrote this book to provide developers with a simple and inviting way to learn about gulp and the tools that are needed to use it. My goal is to keep the content simple and to remain aware of the intimidations that I experienced while learning gulp myself. With this in mind, I wanted to create content that never assumed too much from the reader, but also kept a steady pace for a more experienced reader to keep them engaged and ensure they learn the concepts actively.

What this book covers

Chapter 1, Introducing Gulp, focuses on helping you understand the languages and tools that you will use. You will learn how to use gulp to perform automated tasks for your development projects.

Chapter 2, Getting Started, focuses on getting your local environment set up by installing any software that is needed to move forward. You will learn how to use a command-line interface and take a look at the anatomy of a gulpfile.

Chapter 3, Performing Tasks with Gulp, covers how to create a set of base tasks that you will build upon in the following chapters. These base tasks include concatenation, minification, and preprocessing of your project files.

Chapter 4, Using Node.js Modules for Advanced Tasks, explores when and why to use node.js modules instead of gulp plugins in our tasks. You will learn how to create new tasks to run a static server, keep your project in sync across devices, and take advantage of node.js' module definitions in your client-side code.

Chapter 5, Resolving Issues, covers how to improve your tasks by adding better error handling, ordering your source files, and cleaning up your compiled code. Additionally, you will learn how to set up task dependencies, generate source maps, and use an external configuration file.

What you need for this book

To follow the instructions in this book, you will need to have a computer running Mac OS X, Linux, or Windows and a code editor, such as Sublime Text or Textmate. You should also have a basic understanding of how to build websites using HTML, CSS, and JavaScript. This book will build on top of these skills and teach you ways to use them to improve your development workflow.

Who this book is for

If you are a developer who is new to build systems and task runners, but have had prior experience with web development and have basic knowledge of HTML, CSS, and JavaScript, this is the book for you. It will guide you through the process of using gulp to automate several common development tasks so that they can save time and focus on what is most important—writing great code!

Conventions

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

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "In the code, we have added our concat reference."

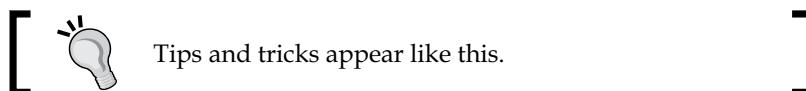
A block of code is set as follows:

```
// Styles Task
gulp.task('styles', function() {
  return gulp.src('app/css/*.css')
    .pipe(plumber())
    .pipe(concat('all.css'))
    .pipe(myth())
    .pipe(gulp.dest('dist'));
});
```

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

```
npm install beeper --save-dev
```

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



Reader feedback

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

To send us general feedback, simply e-mail feedback@packtpub.com, and mention the book's title in the subject of your message.

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

Customer support

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

Downloading the example code

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

This book contains code examples that feature the latest improvements to the gulp project.

Due to the close proximity between the release of this book and the official release of those features, we have provided an additional code bundle that features examples using version 3.8.

Errata

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

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

Piracy

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

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

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

Questions

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

1

Introducing Gulp

Development always starts off in a simple way. You come up with a great idea and then plan out how to build it. Quickly, you scaffold your project structure and organize everything to perfection. As you progress, your small idea starts to grow into a much larger application. You soon realize that your project has become heavy and bloated, and to remedy this, you perform a series of little mundane operations each time you modify your code to keep it small and efficient. Suddenly, all of these repetitive tasks seem to pull you down at the height of your coding victory! You tell yourself that there must be a better way.

The good news is that you are absolutely right. The solution to this development obstacle lies in utilizing build systems. Build systems are some of the most valuable tools in a developer's toolbox, and if you've never used one before, you're soon going to wonder how you ever worked without one.

In software development, build systems such as Make were initially used to compile code into executable formats for use in an operating system. However, in web development, we have a completely different set of practices and operations to contend with. Over the past few years, the growth of the Web has led to an increasing interest in using build systems to more capably handle the growing complexities of our applications and project workflows.

As developers, it is important for us to anticipate these growing complexities. We must do all that we can to improve our workflows so that we can build efficient projects that allow us to focus on what we do best: write great code.

In this book, we are going to explore gulp, one of the most popular JavaScript build systems available today. Instead of dropping you right into the code—abandoning you to sink or swim on your own—we will break apart the learning process into simple, understandable chunks that can be easily consumed and referenced if you get hung up at any point. All that you need to follow the instructions in this book is a general understanding of web development and how to write basic HTML, CSS, and JavaScript.

The first step toward using build systems is quite often viewed as the most intimidating, and understandably so. For years, I viewed the command line as a tool that was only beneficial to programmers and system administrators. I even resisted learning about node.js because I feared the amount of time and dedication required to study it would be greater than how much I could actually benefit from it.

These feelings of intimidation and resistance are completely normal and are felt by many developers just like you. We are overwhelmingly exposed to new tools and frameworks on a daily basis. It is our responsibility as developers to evaluate these tools to determine their overall value based on the time investment required to implement them into our projects. When it comes to some tools, developers simply don't dig deep enough to identify the parts that might be useful to them.

I've come to realize that these things aren't as complicated as we sometimes make them, but many developers are still psyching themselves out before they even really get started. It's important to remember that these tools are not too complicated or large for you to learn and use within a reasonable amount of time. They may be foreign at first, but they are not beyond your grasp and understanding.

What is gulp?

Gulp is a streaming JavaScript build system built with node.js; it leverages the power of streams and code-over-configuration to automate, organize, and run development tasks very quickly and efficiently. By simply creating a small file of instructions, gulp can perform just about any development task you can think of.

Gulp uses small, single-purpose plugins to modify and process your project files. Additionally, you can chain, or pipe, these plugins together into more complex actions with full control of the order in which those actions take place.

Gulp isn't alone though; it is built upon two of the most powerful tools available in the development industry today: node.js and npm. These tools help gulp perform and organize all of the wonderful things that it empowers us to do.

What is node.js?

Node.js, commonly referred to as node, is a powerful JavaScript platform that is built on top of Google Chrome's JavaScript runtime engine, V8. This gives us the ability to write JavaScript code on a server, or in our case, on our local machine. Using node, we now have the opportunity to write both the backend and frontend of a web application entirely in JavaScript. For the purposes of this book, we will only be using it as a means to run local tooling applications.

Node.js ships with npm, a companion package manager that facilitates the installation, storage, and creation of modular components that you can use to create applications. Together, these two tools are the engine behind how gulp operates and organizes its plugin ecosystem.

As I mentioned in the introduction, new tools such as node.js can bring about overwhelming thoughts or feelings of intimidation. This is especially true for those who focus entirely on the frontend side of development. However, when it comes to frontend, often the hardest part is just convincing yourself to get started. Sometimes, all you need is a simple project that can help build your confidence. In the following chapters, this is exactly what we are going to focus on, and soon all of that intimidation will melt away.

Why use gulp?

There are many uses for gulp, but as a newcomer, it might be difficult for you to identify how you can use it to improve your project workflow. With the ever-growing number of tools and frameworks, it becomes difficult to set aside enough time to research and choose the right one for your project or team. To better understand the benefits of gulp, let's identify a few of the defining reasons why to use it and what sets it apart from similar tools.



Project automation

First and foremost, the ability to automate your workflow is incredibly valuable. It brings order to the chaotic amount of tasks that need to be run throughout development.

Let's imagine that you recently developed a big application, but instead of being able to allow the necessary time to put together a proper build system, you were pressured into completing it within an incredibly short timeframe.

Here's an example of this: For the past few days, your boss has been gathering feedback from users who claim that slow load times and performance issues are preventing them from getting their work done and damaging their user experience. It has become so frustrating that they have even threatened to move to another competing service if the performance doesn't improve soon.

Due to the short deadline, the sacrifices that were made during development have actually caused problems for your users, and the maintenance needed to resolve those problems has now become a large burden on you and your team.

Naturally, your boss is rather upset and demands that you figure out a way to correct these issues and deliver a more performant service. Not only that, your boss also expects you to have a sustainable solution so you can provide this across all of your team's future projects as well. It's quite a burden, especially with such short notice. This is a perfect example of where gulp can really save the day.

To deliver better load times in your application, you would need to compress your overall file sizes, optimize your images, and eliminate any unnecessary HTTP requests.

You could implement a step in your workflow to handle each of these manually, but the problem is that workflows often flow forward and backward. No one is infallible, and we all make mistakes. A big part of our job is to correct our mistakes and fix bugs, which requires us to take a step back to resolve any issues we run into during development.

If we were to plan out a step in our workflow to handle these items manually, it would become a huge burden that would most likely take up much of our time. The only practical way to handle optimizations like these is to automate them as an ongoing workflow step. Whether we are just starting, finishing up, or returning to our code to fix bugs, our optimizations will be handled for us.

While things like these should usually be part of your initial project setup, even as an afterthought, gulp makes resolving these issues incredibly easy. Also, it will set you up with a solid base that you can include in future projects.

There are many additional tasks that we can add to our list of automations. These include tasks such as CSS preprocessing, running an HTML server, and automatically refreshing your browser window upon any changes to your code. We will be covering all of those and more in the upcoming chapters.

Streams

At the heart of gulp is something called streams, and this is what sets it apart from other JavaScript build systems. Streams were originally introduced in Unix as a way to "pipe" together small, single-purpose applications to perform complex, flexible operations. Additionally, streams were created to operate on data without the need to buffer the entire file, leading to quicker processing. Piping these small applications together is what is referred to as a pipechain. This is one of the core components of how we will organize and structure our tasks in gulp.

Like Unix, node.js has its own built-in stream module. This stream module is what gulp uses to operate on your data and perform tasks. This allows developers to create small gulp plugins or node modules that perform single operations and then pipe them together with others to perform an entire chain of actions on your data. This gives you full control over how your data is processed by allowing you to customize your pipechain and specify how and in what order your data will be modified.

Code over config

Another reason why many developers find gulp to be a more natural alternative to other JavaScript build systems is because the build file you create is written in code, not config. This may be a matter of personal preference, but I know that this was a fundamental reason why I chose to use gulp over other build systems.

As I mentioned before, by learning more about gulp, you are also learning the basics of node.js, simply because you're writing code for a node.js application. With a build system that uses a config file, you're missing out on the value of learning the core code syntax of the platform you are using.

Summary

In this chapter, we learned about the importance of build systems in software development and the growth of interest for their usage in modern web development workflows.

As we introduce new tools such as preprocessors and JavaScript libraries, we should have a way to properly organize those files into an efficient workflow and build them for production-ready releases.

We discussed the tools that we will be using throughout the rest of the book and how they all work together and interact with one another to provide us with a solid build system solution that we can use for our projects.

With a basic understanding of these tools and their uses, we can now begin to learn how to set up our local environment for gulp. In the upcoming chapter, we will learn about our command-line application, install our software, and prepare our project to begin writing code.

2

Getting Started

Before we dive into gulp, we need to cover some basic information to make sure we get started at the right pace. The most common reason why people end up avoiding build systems such as gulp is because they have a preconceived idea that the command line is inherently hard and complicated. I know this because I've been there myself. Once I got over my initial hesitation and decided to dedicate some time to understand the command line, I've been a much happier developer, and I'm sure you will be too.

In addition to learning how to use the command line, we are also going to understand the installation of node.js and npm. These two tools allow us to run gulp and manage the gulp plugins that we will be using in our project.

Finally, we will cover the basics of using npm, and we will use it to install gulp. This will provide you with all of the necessary knowledge to get comfortable with using the command line to install packages.

Getting comfortable with the command line

Your computer's command-line is one of the most powerful tools in your development toolset. If you've never used a command-line or if you're still wondering what it even is, don't worry. We are going to take a look at some common commands and patterns that you will use to complete the rest of the book and set you up for future command-line usage.

First, we need to discuss the differences between operating systems and their command-line interfaces that we will use. We are going to specifically cover two topics: Terminal on Mac/Linux and PowerShell on Windows.

Mac OS X Terminal: If you're using a Mac (or Linux), Terminal comes preinstalled as a system application. Simply search for it and run the application, and you're all set.

Windows PowerShell: Our Windows setup is a bit different in that we will use an additional application as our command-line interface. Windows PowerShell is a powerful application that will make your command-line experience much more enjoyable and consistent with the commands that we will run on Mac. Windows ships with a more basic command-line interface called Command Prompt; however, due to the differences in the syntax and added features, PowerShell is the clear choice when doing any command-line operations on Windows.

If you're running Windows 8, PowerShell comes preinstalled. However, if you're running anything below Windows 8, it will require an additional download.



To download PowerShell, visit <http://technet.microsoft.com/en-us/library/hh847837.aspx>.



Once you have your command-line interface installed and running, take a moment to understand what you see on the screen. When you first open your command-line, you will most likely be greeted with something that appears completely alien to you. It should look something like this on Mac and Linux:



On Mac, the displayed text should look something like this:

```
travs-macbook:~ travis$
```

On Windows, it should look something like this:

```
PS C:\Users\Travis>
```

This is a visual reference to our current location in our filesystem. The reason why these lines are different is because these operating systems have different filesystems. The good news is that you won't have to worry much about this. The important thing is to understand what you're seeing because this will soon change as we use commands and will also help you visualize where you are within the filesystem.

At the end of these lines, you will notice a flashing cursor. This cursor verifies that our command line is ready to accept commands. Every command we will use shares a common syntax that is used to run our commands and specify what we would like each command to do. A common structure of this is as follows:



The command is as follows:

```
application action -flag arguments
```

As you can see, each parameter we use in a command is separated by a single space to separate the application from its actions, flags, and arguments.

In the command application is the first thing listed in each command followed by the action we would like it to perform. In some cases, the only purpose of an application is to provide an action, so the additional action parameter doesn't apply in those situations.

Next, we provide something called flag. Flags are optional and always preceded by a single dash such as -v. They are used to specify additional options for the action or application.

Finally, we list our arguments that are the custom input we provide to our application. Arguments will usually contain a name or set of names to either create or reference files depending on the application you are using.

In most cases, this order will remain consistent. As long as each parameter is separated by a single space, everything should work as expected. However, some applications require a specific order to execute the commands correctly. Be sure to check out the documentation of the application if you run into any problems.



Most, if not all, command-line applications feature built-in help documentation in case you get stuck. To access this, simply pass in a -help or -h flag when running the application. The documentation will be displayed right inside of your command-line application interface.

Command reference

While learning how to use the command-line, it is often easy to forget the basic commands that you will need to use. So, I've created this simple reference. Go over each of the standard commands that we will be using throughout the book.

We'll start off with the basics, and then I'll show you some neat shortcuts that you can use while setting up your project structure. The commands we are going to cover are ls, cd, mkdir, touch, and ni.

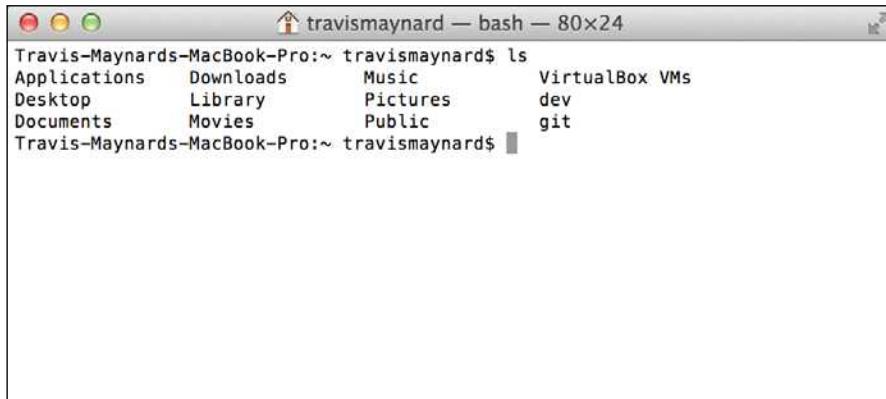
Listing files and folders (ls)

The ls command will allow us to see what our current directory contains. You will use this a lot to see what is inside of your folders and ensure you are in the right location in your file system.

For listing files and folders, use the following command:

```
ls
```

A screenshot listing files and folders is as follows:



The screenshot shows a terminal window titled "travismaynard — bash — 80x24". The command "ls" is run, displaying the contents of the user's home directory (~). The output shows several folders: Applications, Downloads, Music, VirtualBox, Desktop, Library, Pictures, dev, Documents, Movies, Public, and git. The prompt "Travis-Maynards-MacBook-Pro:~ travismaynard\$" is visible at the bottom.

Changing directory/folder (cd)

The `cd` command stands for "change directory." It allows you to navigate through your file system. It will accept both a path relative to the directory you are currently in and an absolute path to navigate directly to any directory in the file system.

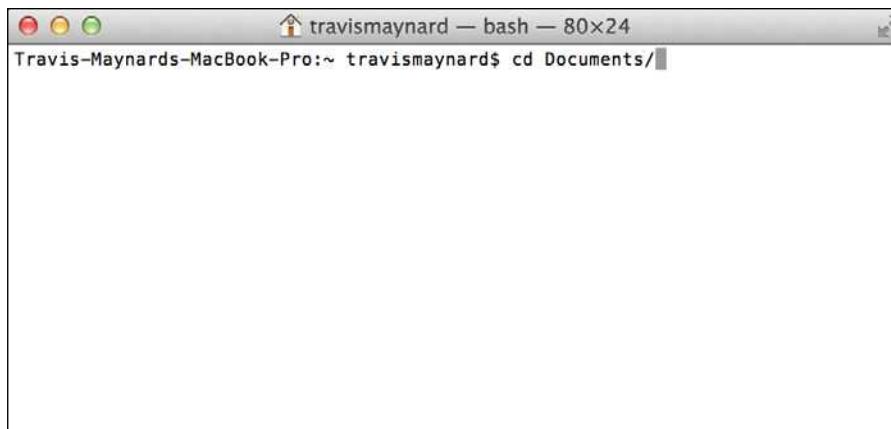
The command for relative paths is as follows:

```
cd folder  
cd folder/sub-folder
```

The command for an absolute path is as follows:

```
cd /users/travis/documents
```

The next screenshot gives the picture of how to change your directory:

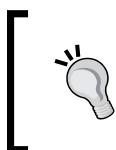


The screenshot shows a terminal window titled "travismaynard — bash — 80x24". The command "cd Documents/" is typed into the terminal. The prompt "Travis-Maynards-MacBook-Pro:~ travismaynard\$ cd Documents/" is visible at the bottom.

To navigate out of a folder, you simply pass in `cd ..` in place of a path/folder. This will take you up one level in your tree.

To move up one tree level, use the following command:

```
cd ..
```



While typing out your path or folder names, you can use the *Tab* key to autocomplete the directory names so that you don't have to type out the full path. It's a great little shortcut that can save you a lot of time when navigating around your file system.



Making a directory/folder (`mkdir`)

The `mkdir` command stands for "make directory." It allows you to create folders. Additionally, as a shortcut, you can pass in multiple folder names separated by a single space to create multiple folders at the same time.

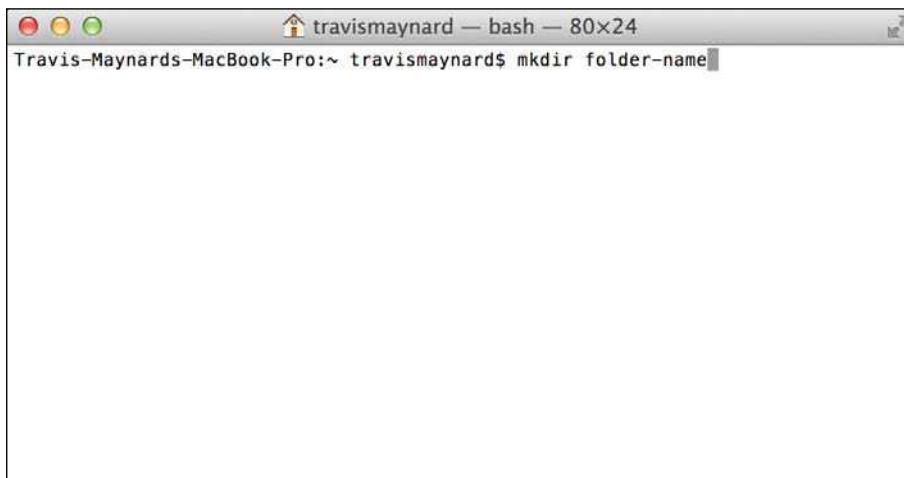
To create a single folder, use the following command:

```
mkdir folder-name
```

To create multiple folders, use the following command:

```
mkdir folder-one folder-one/sub-folder folder-two
```

The next screenshot shows how to make a directory:



Creating a file on Mac/Linux (touch)

The `touch` command is actually used to change a file's timestamps. However, if you pass it a filename that does not exist yet, it will create a blank new file for you on Mac/Linux systems. This is why you will see it often used as a way to create new files. Like `mkdir`, you can also pass in multiple filenames separated by a single space to create multiple files at once.

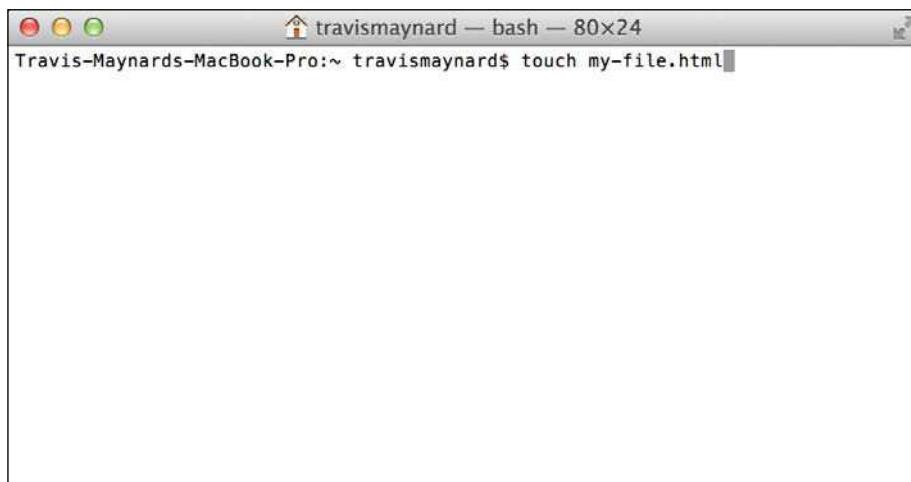
For creating a file on Mac/Linux, use the following command:

```
touch my-file.html
```

For creating multiple files on Mac/Linux, use the following command:

```
touch my-file.html styles.css
```

The following screenshot shows how to create a new file:



Creating a file on Windows (ni)

The `ni` command stands for "new item." It allows you to create a new file on Windows systems using PowerShell.

For creating a file on Windows, use the following command:

```
ni my-file.html -type file
```

Unlike other commands, we are required to specify a flag with the type of item that we wish to create. When using this command, we are required to specify the type of item because `ni` can be used to create folders as well.



For this book, we will only create folders with the `mkdir` command, but feel free to use `ni` if you are more comfortable with it.



If you do not provide the application with the flag, then it will prompt you to input the information before you can continue with the application.

Administrator permissions (`sudo`)

On Mac and Linux systems, you may run into permission issues as you run some of your commands, especially if the commands are written to protected areas of your file system. This is put in place to protect you from accidentally overwriting or deleting important files. In the case where you actually intend to create or modify files in protected areas, you will need to add this `sudo` keyword to the beginning of your commands.

For creating a folder with administrator permission, use the following command:

```
sudo mkdir folder-name
```

For creating a file with administrator permission, use the following command:

```
sudo touch my-file.html
```

By adding the `sudo` keyword to your commands, the system will prompt for your administrator password on the next line of your command-line application. If you enter your password correctly, the command will run with full administrator access and override any permission restrictions. Otherwise, you will receive permission errors and the command will halt.

By examining these commands, you can quickly notice the common pattern they all share. Having a familiarity with this shared pattern is great because all of the new commands we learn throughout the book will follow it as well.

Creating your project structure

Having learned all of these great new commands, we're now going to use them to scaffold our project folder. First, let's make sure we're all in the same starting directory. For this, use the following command:

```
cd ~
```



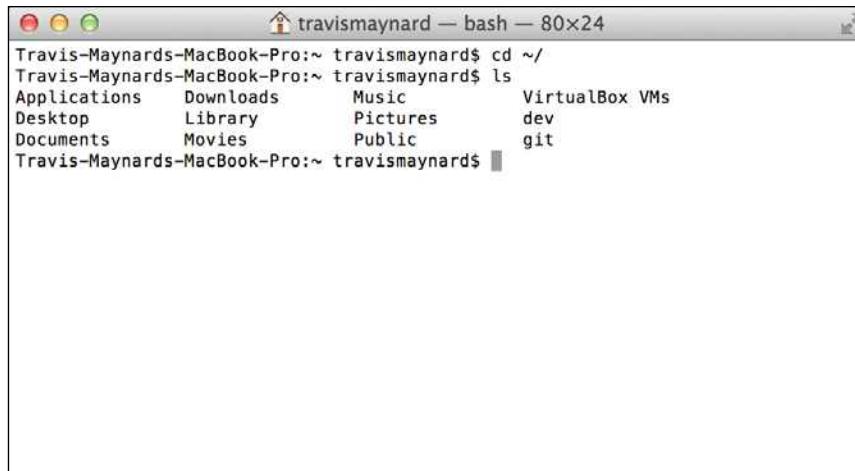
The ~ is a nice little shortcut for our user's home directory, which is a shortcut for /Users/Username.



Next, we're going to list out all of the files and folders in this directory to get a quick look at what it contains and ensure we are where we want to be. For listing files and folders, use the following command:

ls

Once you've run this command, your terminal window will respond with a listing of all your files and folders inside the current directory, which is shown as follows:



```
travismaynard$ cd ~/
travismaynard$ ls
Applications    Downloads    Music        VirtualBox VMs
Desktop        Library     Pictures      dev
Documents       Movies      Public       git
Travis-Maynards-MacBook-Pro:~ travismaynard$
```



On Mac, if you would like to receive more detailed information, simply add a space followed by the -alht flag.

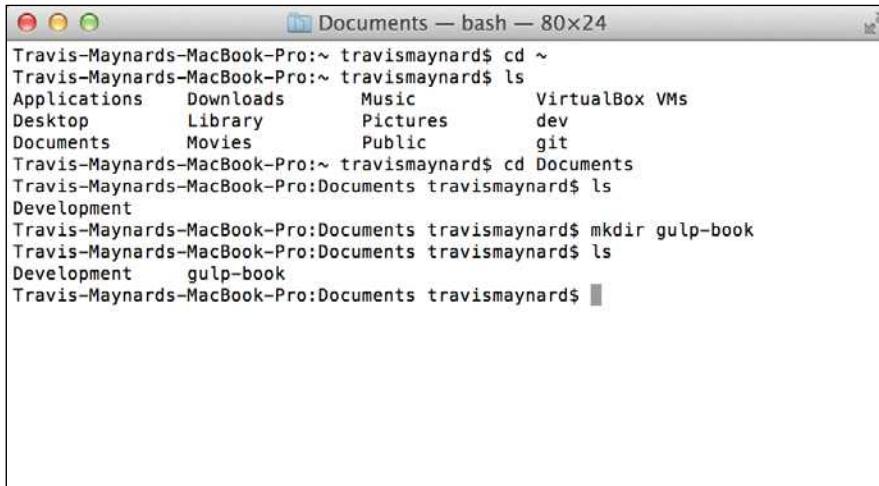


Next, we're going to create a new folder named `gulp-book` for our gulp project to live in. If you would like to create this folder in another directory, now is the time to put your `cd` and `ls` commands to good use. Once you have navigated to a directory you are comfortable with, it's time to create your new project folder, which is done using the following command:

mkdir gulp-book

Getting Started

Once you run this command on your terminal window, a new folder named `gulp-book` will be created, which is shown as follows:

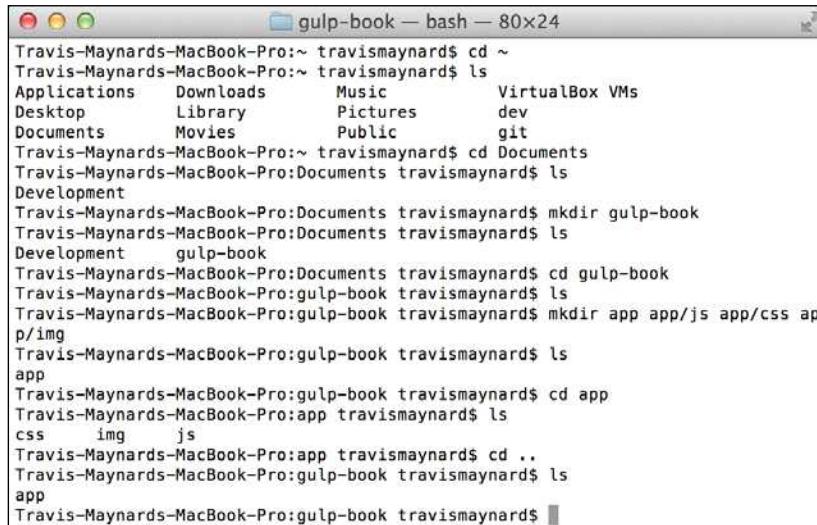


```
Travis-Maynards-MacBook-Pro:~ travismaynard$ cd ~
Travis-Maynards-MacBook-Pro:~ travismaynard$ ls
Applications Downloads Music VirtualBox VMs
Desktop Library Pictures dev
Documents Movies Public git
Travis-Maynards-MacBook-Pro:~ travismaynard$ cd Documents
Travis-Maynards-MacBook-Pro:Documents travismaynard$ ls
Development
Travis-Maynards-MacBook-Pro:Documents travismaynard$ mkdir gulp-book
Travis-Maynards-MacBook-Pro:Documents travismaynard$ ls
Development gulp-book
Travis-Maynards-MacBook-Pro:Documents travismaynard$
```

Next, we need to move into that directory so we can scaffold out the rest of our project structure. Instead of creating a single folder at a time, we will pass in the remaining folders we need to create all at once, which can be done using the following command:

```
cd gulp-book
mkdir app app/js app/css app/img
```

The next screenshot shows the creation of multiple directories:



```
Travis-Maynards-MacBook-Pro:~ travismaynard$ cd ~
Travis-Maynards-MacBook-Pro:~ travismaynard$ ls
Applications Downloads Music VirtualBox VMs
Desktop Library Pictures dev
Documents Movies Public git
Travis-Maynards-MacBook-Pro:~ travismaynard$ cd Documents
Travis-Maynards-MacBook-Pro:Documents travismaynard$ ls
Development
Travis-Maynards-MacBook-Pro:Documents travismaynard$ mkdir gulp-book
Travis-Maynards-MacBook-Pro:Documents travismaynard$ ls
Development gulp-book
Travis-Maynards-MacBook-Pro:Documents travismaynard$ cd gulp-book
Travis-Maynards-MacBook-Pro:gulp-book travismaynard$ ls
Travis-Maynards-MacBook-Pro:gulp-book travismaynard$ mkdir app app/js app/css app/img
Travis-Maynards-MacBook-Pro:gulp-book travismaynard$ ls app
Travis-Maynards-MacBook-Pro:gulp-book travismaynard$ cd app
Travis-Maynards-MacBook-Pro:app travismaynard$ ls
css img js
Travis-Maynards-MacBook-Pro:app travismaynard$ cd ..
Travis-Maynards-MacBook-Pro:gulp-book travismaynard$ ls app
Travis-Maynards-MacBook-Pro:gulp-book travismaynard$
```

The preceding command has created an `app` folder and three subfolders within it named `css`, `img`, and `js`. All of our folders are created and ready for us to add them to our files.

For Mac/Linux Terminal, use the following command:

```
touch index.html
```

For Windows PowerShell, use the following command:

```
ni index.html -type file
```

With these commands, we've simply created a blank `index.html` file in our base directory. Now, let's create a `gulpfile`.

For creating a file on Mac/Linux Terminal, use the following command:

```
touch gulpfile.js
```

For Windows PowerShell, use the following command:

```
ni gulpfile.js -type file
```

The screenshot of the preceding command is as follows:

A `gulpfile` is a set of instructions that `gulp` uses to run your tasks. All the code that we will be writing for `gulp` will be contained in this file. We will be coming back to this file very soon.

Getting Started

Hopefully, this is all starting to feel familiar. The more you use it, the more comfortable you'll be and the quicker you will be able to execute commands.

We've created some files in our base directory, but now we need to create some blank files in our app directories. Next, let's create a couple of blank CSS and JavaScript files for later use.

For Mac/Linux Terminal, use the following command:

```
touch app/css/main.css app/css/secondary.css app/js/main.js app/js/
secondary.js
```

When using Terminal, we can create multiple files at once, much like our `mkdir` command from earlier.

For Windows PowerShell, use the following command:

```
ni app/css/main.css -type file
ni app/css/secondary.css -type file
ni app/js/main.js -type file
ni app/js/secondary.js -type file
```

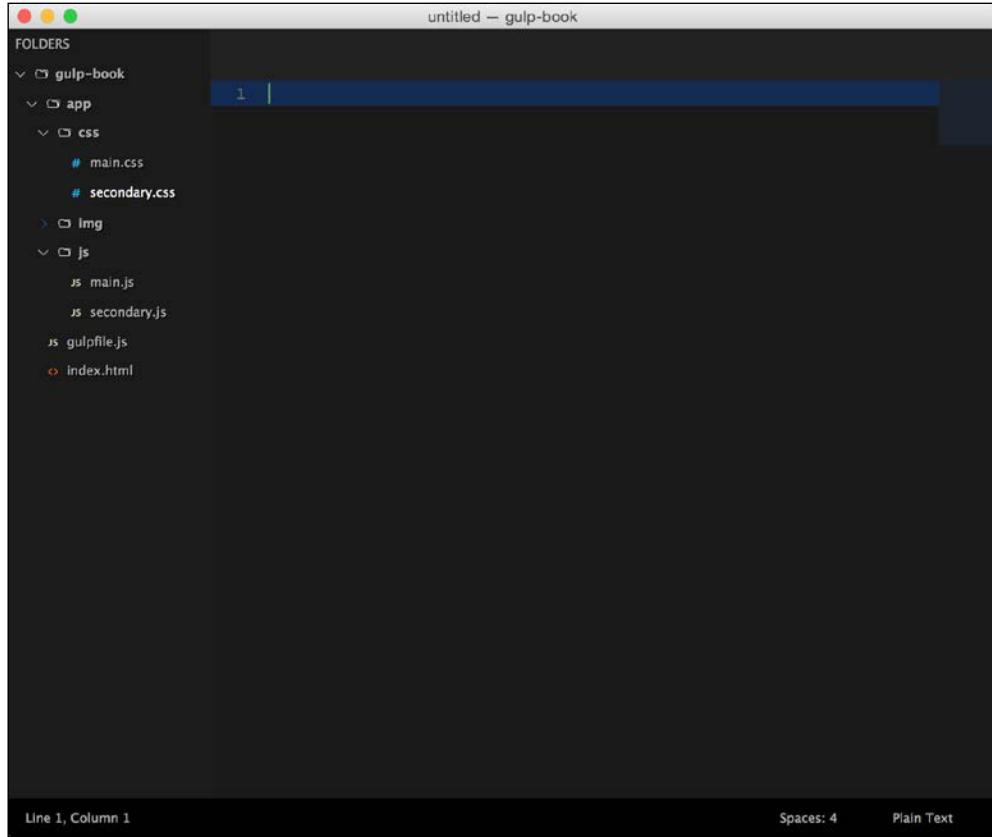


```
Travis-Maynards-MacBook-Pro:gulp-book travismaynard$ touch app/css/main.css app/
css/secondary.css app/js/main.js app/js/secondary.js
Travis-Maynards-MacBook-Pro:gulp-book travismaynard$ ls
app          gulpfile.js      index.html
Travis-Maynards-MacBook-Pro:gulp-book travismaynard$ cd app/
Travis-Maynards-MacBook-Pro:app travismaynard$ ls
css    img    js
Travis-Maynards-MacBook-Pro:app travismaynard$ cd css
Travis-Maynards-MacBook-Pro:css travismaynard$ ls
main.css    secondary.css
Travis-Maynards-MacBook-Pro:css travismaynard$
```

If all went well, then you should be all set! Just to make sure, load your project folder into your favorite code editor or integrated development environment. Your tree should look like this:

```
gulp-book/
  -app
    -css
      - main.css
      - secondary.css
    -img
    -js
      - main.js
      - secondary.js
  index.html
  gulpfile.js
```

The next screenshot shows the tree structure:



If your tree looks like this, then great! You've successfully learned how to use your command line to scaffold a basic project.

If your tree looks different, then you can take a moment to revisit the commands and try again, or you can patch up the missing files and folders in your code editor. It's up to you.

In the upcoming chapters, we will use this base project structure to build a small website in order to demonstrate gulp's capabilities. But first, let's install the remaining software we will be using throughout the book.

Adding content to the project

After scaffolding our project folders and files, we must add code to our project. Our project is going to be a rather simple one-page HTML website. However, by setting this code up together, it will help us demonstrate the work that is taking place as we run our gulp tasks in the upcoming chapters.

Keep in mind that these examples will be rather simple only to reinforce those demonstrations. You are more than welcome to add in any additional code that you would like, but for the sake of simplicity and clarity, the code examples in this book are designed specifically to demonstrate the work our tasks will do to our code.



Downloading the example code

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

Preparing our HTML file

For our `index.html` file, we just need to provide a basic structure and link it up to the distributable files in our head. The code is as follows:

```
<!doctype html>
<html lang="en">
<head>
    <title>Gulp Book Project</title>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, height=device-height, initial-scale=1.0, maximum-scale=1.0, user-scalable=no" />
    <meta name="description" content="A simple one-page website to demonstrate
        gulp." />
    <meta name="keywords" content="gulp, book, example" />
    <meta name="robots" content="index, follow" />
    <link rel="stylesheet" href="dist/all.css" />
```

```
</head>

<body>
  <div id="core">
    <div class="box">
      
      <h1>Gulp Book Example</h1>
      <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit.  
Fugiat atque unde  
doloremque illo, voluptatibus repellendus iusto, praesentium  
officia necessitatibus  
consectetur blanditiis neque eveniet accusamus dolorum labore  
iure vel,  
tempora odio.</p>
      <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit.  
Reiciendis dignissimos  
commodi minus sint animi itaque praesentium, natus vel eaque,  
molestias sequi  
modi quaerat aliquam in, quisquam quos, impedit maiores  
ratione!</p>
      <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit.  
Quos ut possimus,  
repellat vero modi, aliquam pariatur deserunt voluptas quam  
omnis maiores eveniet  
quo ipsam totam quasi recusandae, rem sit deleniti.</p>
    </div>
  </div>
  <script src="dist/all.js"></script>
</body>
</html>
```

Preparing our CSS

Once our HTML has been set up, we should begin writing our CSS. For the purposes of this example, we are going to keep the files rather small just to demonstrate the work that will be occurring when we run our tasks. However, if you feel comfortable up to this point, feel free to use your own code.

Let's open our `main.css` file that is located in our project's `css` directory. Inside this file, type or paste in the following code:

```
/* Myth Variables */
:root {
  /* Colors */
  --red: #F05D5D;
  --green: #59C946;
```

```
--blue: #6F7AF1;
--white: #FFFFFF;
--grey: #EEEEEE;
--black: #000000;
}
body {font:300 16px sans-serif; background:var(--grey);}
```

This code may seem a bit unfamiliar to you, especially if you're used to another CSS preprocessor like Sass. However, the ideas are the same; only the syntax has changed. Myth is a more pure way to add in variables and calculations to your CSS files as it is based on the official CSS spec that determines what features will be added in the future. Myth's only shortcoming to other preprocessors is its lack of features.

Next, let's open our `secondary.css` file that is located in the same directory. In this file, we will add some additional styles that we will soon join together with our `gulp-concat` plugin. The code for the `secondary.css` file is as follows:

```
#header {padding:2em; background: var(--blue);}
#core {width:80%; max-width:900px; margin:0 auto;}
#footer {padding:2em; background: var(--red);}

.box {padding:2em; background:var(--white);}
.gulp-logo {width:125px; margin:0 auto; display:block;}
```

We now have two separate CSS files that set preprocessor variables and then use those variables to output colors. The task we wrote in the preceding code will ensure that both of these files will be joined together. They will then be preprocessed so that all of these new variables we have created will properly be assigned to the elements where we have assigned them in our CSS.

Preparing our JavaScript

Next, we need to add some code to our JavaScript files. Again, for the sake of succinctness and clarity, we will keep these examples quite small just to demonstrate what is happening to our files when gulp runs.

First, let's open the `main.js` file inside the `js` directory in our main project folder. In this file, we are going to add a line of code to just fire off some basic logs to our browser console. The code is as follows:

```
console.log("I'm logging from the main.js file.");
```

Next, let's open up the `secondary.js` file and add in another log to our browser console, which is as follows:

```
console.log("I'm logging from the secondary.js file.");
```

Our project files have now been set up to demonstrate the processing that we will be doing with gulp. We will be revisiting these files in the next chapter when we begin writing our tasks. Next, we will install the remaining software that we will use throughout the book.

Adding images

Our project will also contain some images. You are more than welcome to include any images you would like; however, if you would like to follow along with the book, I have provided some images for you in the project files for this chapter. You can download these files from Packt's website.

Installing node.js and npm

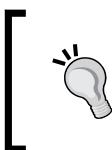
As you learned in the introduction, node.js and npm are the engines that work behind the scenes that allow us to operate gulp and keep track of any plugins we decide to use.

Downloading and installing node.js

For Mac and Windows, the installation is quite simple. All you need to do is navigate over to <http://nodejs.org> and click on the big green **INSTALL** button. Once the installer has finished downloading, run the application and it will install both node.js and npm.

For Linux, there are a couple more steps, but don't worry; with your newly acquired command-line skills, it should be as simple as it can be. To install node.js and npm on Linux, you'll need to run the following three commands in Terminal:

```
sudo add-apt-repository ppa:chris-lea/node.js  
sudo apt-get update  
sudo apt-get install nodejs
```



The details of these commands are outside the scope of this book, but just for reference, they add a repository to the list of available packages, update the total list of packages, and then install the application from the repository we added.

Verifying the installation

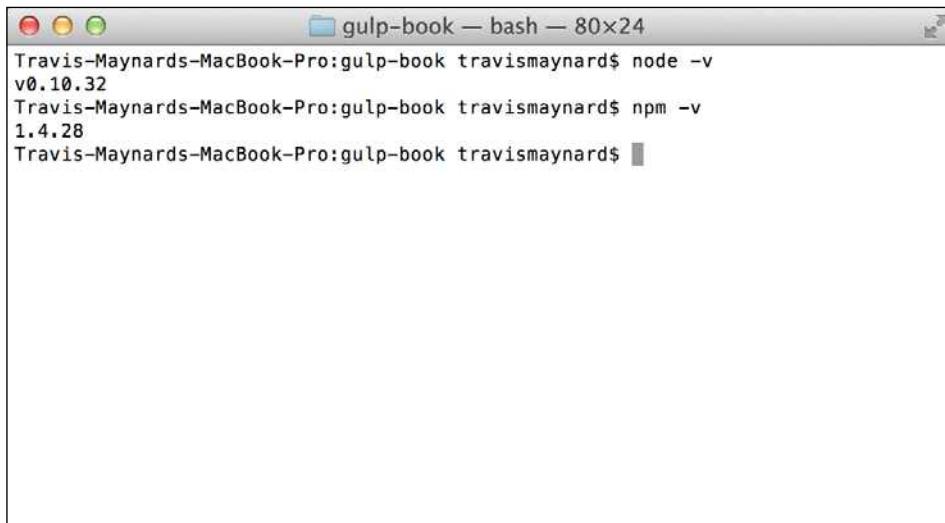
To confirm that our installation was successful, try the following command in your command line:

```
node -v
```

If node.js is successfully installed, `node -v` will output a version number on the next line of your command line. Now, let's do the same with npm:

```
npm -v
```

Like before, if your installation was successful, `npm -v` should output the version number of npm on the next line.



```
gulp-book — bash — 80x24
Travis-Maynards-MacBook-Pro:gulp-book travismaynard$ node -v
v0.10.32
Travis-Maynards-MacBook-Pro:gulp-book travismaynard$ npm -v
1.4.28
Travis-Maynards-MacBook-Pro:gulp-book travismaynard$
```



The `-v` command is a common flag used by most command-line applications to quickly display their version number. This is very useful to debug version issues while using command-line applications.

Creating a package.json file

Having npm in our workflow will make installing packages incredibly easy; however, we should look ahead and establish a way to keep track of all the packages (or dependencies) that we use in our projects. Keeping track of dependencies is very important to keep your workflow consistent across development environments.

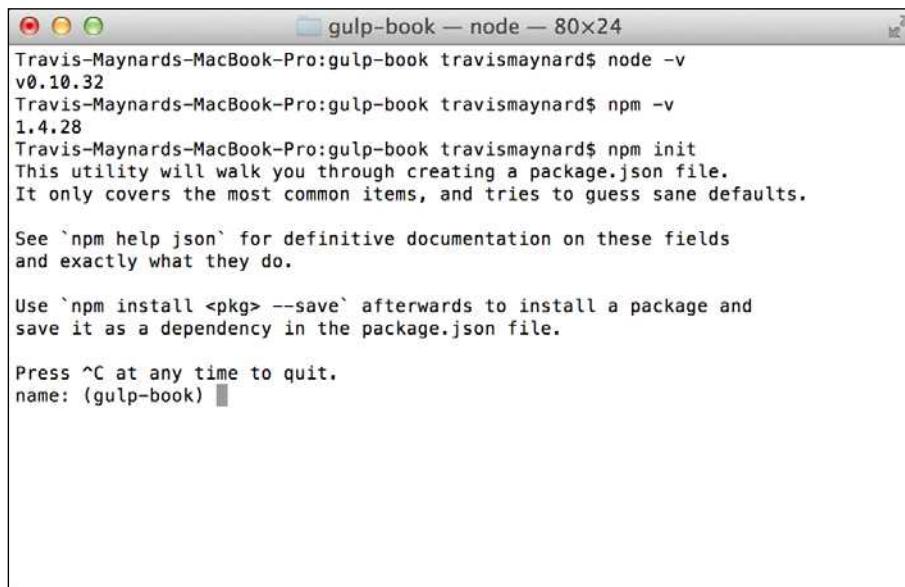
Node.js uses a file named `package.json` to store information about your project, and npm uses this same file to manage all of the package dependencies your project requires to run properly.

In any project using gulp, it is always a great practice to create this file ahead of time so that you can easily populate your dependency list as you are installing packages or plugins.

To create the `package.json` file, we will need to run npm's built in `init` action by using the following command:

```
npm init
```

By using the preceding command, the terminal will show the following output:



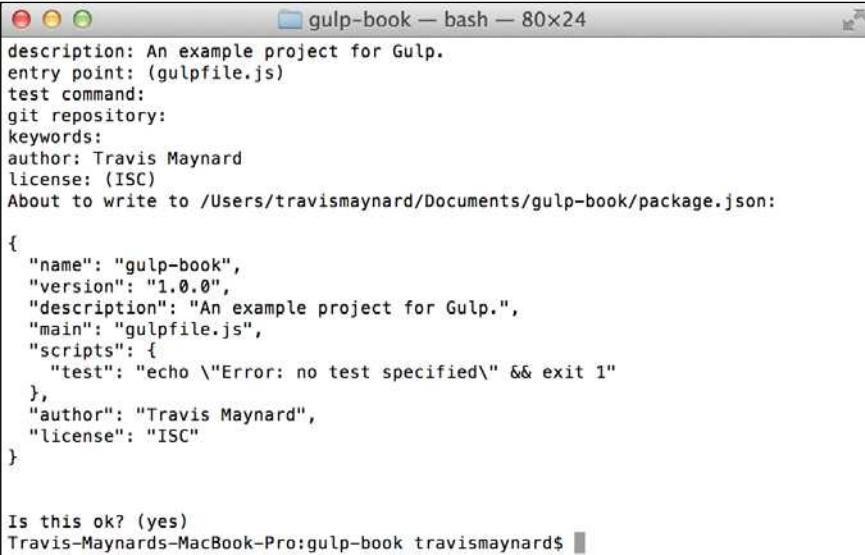
```
gulp-book — node — 80x24
Travis-Maynards-MacBook-Pro:gulp-book travismaynard$ node -v
v0.10.32
Travis-Maynards-MacBook-Pro:gulp-book travismaynard$ npm -v
1.4.28
Travis-Maynards-MacBook-Pro:gulp-book travismaynard$ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sane defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg> --save` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
name: (gulp-book)
```

Your command line will prompt you several times asking for basic information about the project, such as the project name, author, and the version number. You can accept the defaults for these fields by simply pressing the *Enter* key at each prompt. Most of this information is used primarily on the npm website if a developer decides to publish a node.js package. For our purposes, we will just use it to initialize the file so we can properly add our dependencies as we move forward. The screenshot for the preceding command is as follows:



```
gulp-book — bash — 80x24
description: An example project for Gulp.
entry point: (gulpfile.js)
test command:
git repository:
keywords:
author: Travis Maynard
license: (ISC)
About to write to /Users/travismaynard/Documents/gulp-book/package.json:

{
  "name": "gulp-book",
  "version": "1.0.0",
  "description": "An example project for Gulp.",
  "main": "gulpfile.js",
  "scripts": {
    "test": "echo \\\"Error: no test specified\\\" && exit 1"
  },
  "author": "Travis Maynard",
  "license": "ISC"
}

Is this ok? (yes)
Travis-Maynard-MacBook-Pro:gulp-book travismaynard$
```

Installing gulp

With npm installed and our `package.json` file created, we are now ready to begin installing node.js packages. The first and most important package we will install is none other than `gulp` itself.

Locating gulp

Locating and gathering information about node.js packages is very simple, thanks to the npm registry. The npm registry is a companion website that keeps track of all the published node.js modules, including `gulp` and `gulp` plugins. You can find this registry at <http://npmjs.org>. Take a moment to visit the npm registry and do a quick search for `gulp`.

The listing page for each node.js module will give you detailed information on each project, including the author, version number, and dependencies. Additionally, it also features a small snippet of command-line code that you can use to install the package along with readme information that will outline basic usage of the package and other useful information.

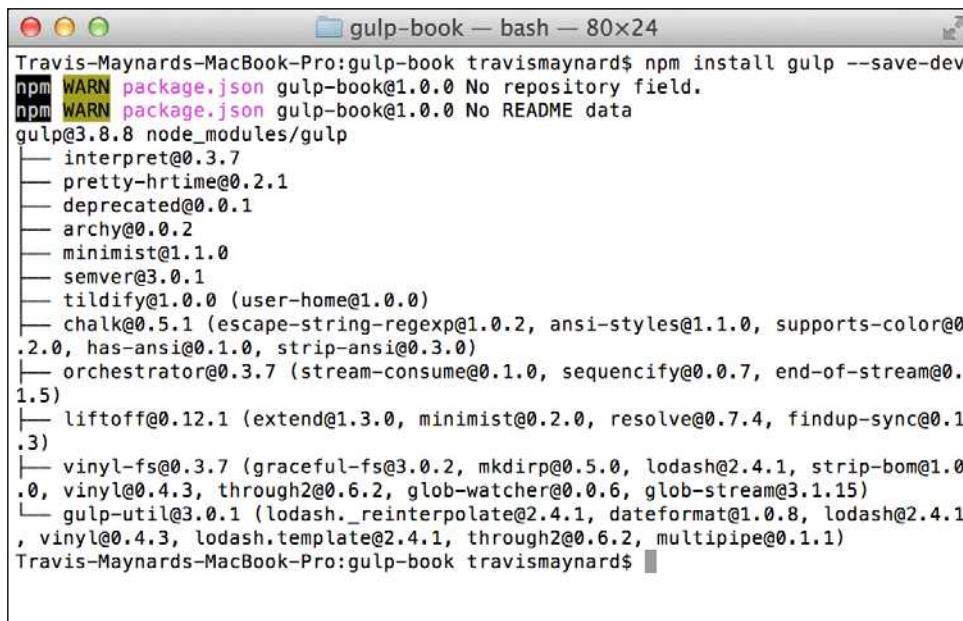
Installing gulp locally

Before we install gulp, make sure you are in your project's root directory, `gulp-book`, using the `cd` and `ls` commands we learned earlier. If you ever need to brush up on any of the standard commands, feel free to take a moment to step back and review as we progress through the book.

To install packages with npm, we will follow a similar pattern to the ones we've used previously. You can also find this small snippet on the gulp page of the npm registry:

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

The preceding command gives the following output on the terminal:



```
Travis-Maynards-MacBook-Pro:gulp-book travismaynard$ npm install gulp --save-dev
npm WARN package.json gulp-book@1.0.0 No repository field.
npm WARN package.json gulp-book@1.0.0 No README data
gulp@3.8.8 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, has-ansi@0.1.0, strip-ansi@0.3.0)
├── orchestrator@0.3.7 (stream-consume@0.1.0, sequencify@0.0.7, end-of-stream@0.1.5)
├── liftoff@0.12.1 (extend@1.3.0, minimist@0.2.0, resolve@0.7.4, findup-sync@0.1.3)
└── vinyl-fs@0.3.7 (graceful-fs@3.0.2, mkdirp@0.5.0, lodash@2.4.1, strip-bom@1.0.0, vinyl@0.4.3, through2@0.6.2, glob-watcher@0.0.6, glob-stream@3.1.15)
  └── gulp-util@3.0.1 (lodash._reinterpolate@2.4.1, dateformat@1.0.8, lodash@2.4.1, vinyl@0.4.3, lodash.template@2.4.1, through2@0.6.2, multipipe@0.1.1)
Travis-Maynards-MacBook-Pro:gulp-book travismaynard$
```

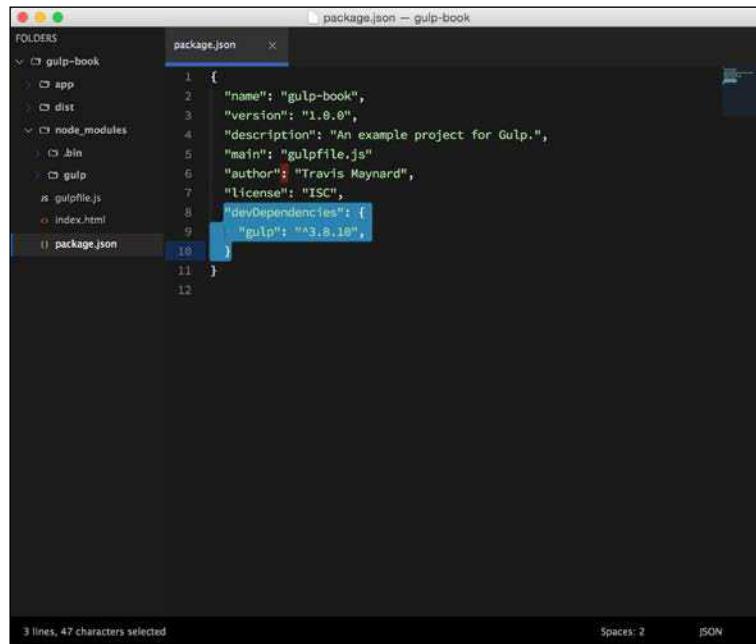
Getting Started

To break this down, let's examine each piece of this command to better understand how npm works:

- **npm**: This is the application we are running.
- **install**: This is the action that we want the program to run. In this case, we are instructing npm to install something in our local folder.
- **gulp**: This is the package we would like to install.
- **--save-dev**: This is a flag that tells npm to add this module to our devDependencies list in our package.json file.

 Additionally, npm has a --save flag that saves the module to the list of devDependencies. These dependency lists are used to separate the modules that a project depends on to run, and the modules a project depends on when in development. Since we are using gulp to assist us in development, we will always use the --save-dev flag throughout the book.

So, this command will use npm to contact the npm registry, and it will install gulp to our local gulp-book directory. After using this command, you will notice that a new folder has been created that is named node_modules. The node_modules folder is where node.js and npm store all of the installed packages and dependencies of your project.



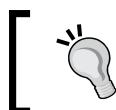
```
package.json
{
  "name": "gulp-book",
  "version": "1.0.0",
  "description": "An example project for Gulp.",
  "main": "gulpfile.js",
  "author": "Travis Maynard",
  "license": "ISC",
  "devDependencies": {
    "gulp": "^3.8.10"
  }
}
```

Installing gulp globally

For many of the packages that we install, this will be all that is needed. With gulp, we must install it once again globally so that we can use gulp as a command-line application from anywhere in our file system in any of the projects that we use. To install gulp globally, use the following command:

```
npm install -g gulp
```

In this command, not much has changed compared to the original command where we installed gulp locally. We've only added a `-g` flag to the command, which instructs npm to install the package globally.



On Windows, your console window should be opened under an administrator account in order to install an npm package globally.



At first, this can be a little confusing, and for many packages it won't apply. Similar build systems actually separate these usages into two different packages that must be installed separately; one that is installed globally for command-line use and another installed locally in your project.

Gulp was created so that both of these usages could be combined into a single package, and, based on where you install it, it could operate in different ways.

Anatomy of a gulpfile

Before we can begin writing tasks, we should take a look at the anatomy and structure of a gulpfile. Examining the code of a gulpfile will allow us to better understand what is happening as we run our tasks.

Gulp started with four main methods: `.task()`, `.src()`, `.watch()`, and `.dest()`. The release of version 4.0 introduced additional methods such as: `.series()` and `.parallel()`. In addition to the gulp API methods, each task will also make use of the node.js `.pipe()` method. This small list of methods is all that is needed to understand how to begin writing basic tasks. They each represent a specific purpose and will act as the building blocks of our gulpfile.

The task() method

The `task()` method is the basic wrapper for which we create our tasks. Its syntax is `.task(string, function)`. It takes two arguments: a string value representing the name of the task and a function that will contain the code you wish to execute upon running that task.

The src() method

The `src()` method is our input, or how we gain access to the source files that we plan on modifying. It accepts either a single string or an array of strings as an argument. The syntax is `.src(string || array)`.

The watch() method

The `watch()` method is used to specifically look for changes in our files. This will allow us to keep gulp running as we code so that we don't need to rerun gulp any time we need to process our tasks. The syntax for this method is `.watch(string, array)`.

The dest() method

The `dest()` method is used to set the output destination of your processed file. Most often, this will be used to output our data into a `build` or `dist` folder that will be either shared as a library or accessed by your application. The syntax for this method is `.dest(string)`.

The pipe() method

The `pipe()` method will allow us to pipe together smaller single-purpose plugins or applications into a pipechain. This is what gives us full control of the order in which we would need to process our files. The syntax for this method is `.pipe(function)`.

The parallel() and series() methods

The `parallel()` and `series()` methods were added in version 4.0 as a way to easily control whether your tasks are ran together - all at once, or in a sequence - one after the other. This is important if one of your tasks requires that other tasks complete before it can be ran successfully.

Understanding these methods will take you far, as these are the core elements of building your tasks. Next, we will need to put these methods together and explain how they all interact with one another to create a gulp task.

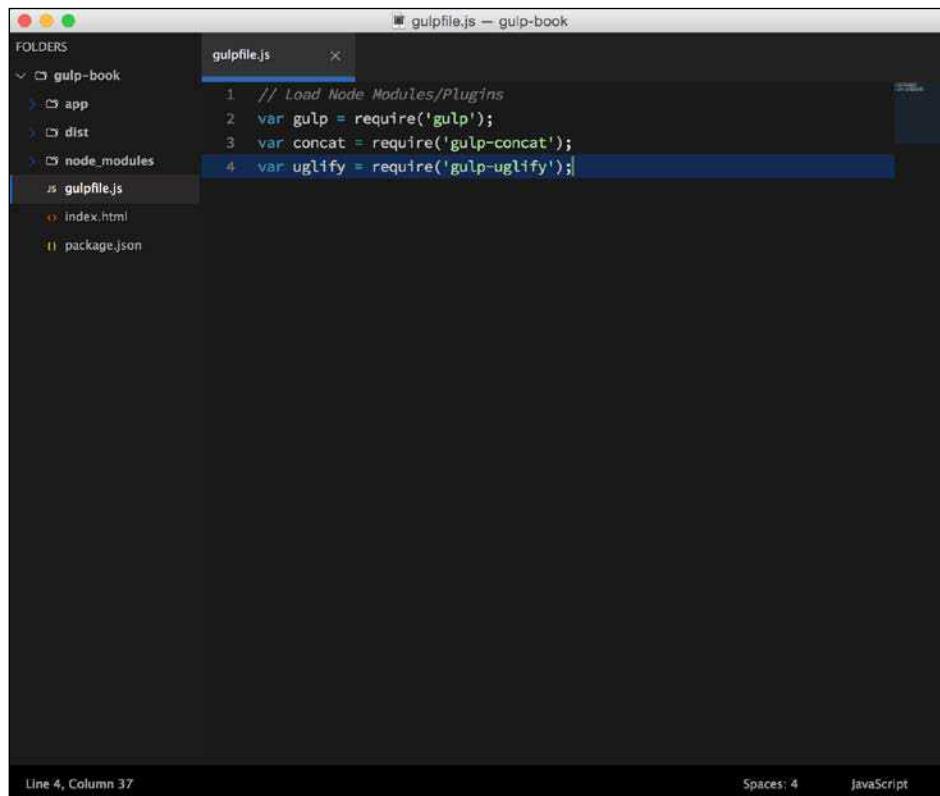
Including modules/plugins

When writing a gulpfile, you will always start by including the modules or plugins you are going to use in your tasks. These can be both gulp plugins or node.js modules, based on what your needs are. Gulp plugins are small node.js applications built for use inside of gulp to provide a single-purpose action that can be chained together to create complex operations for your data. Node.js modules serve a broader purpose and can be used with gulp or independently.

Just remember that you must install each plugin or module using npm prior to including them in your gulpfile. The following is an example of what this code will look like:

```
// Load Node Modules/Plugins
var gulp = require('gulp');
var concat = require('gulp-concat');
var uglify = require('gulp-uglify');
```

The gulpfile.js file will look as shown in the following screenshot:



A screenshot of a code editor window titled "gulpfile.js — gulp-book". The left sidebar shows a file tree with "FOLDERS" containing "gulp-book", "app", "dist", and "node_modules". Inside "node_modules", there is a "gulpfile.js" file and other files like "index.html" and "package.json". The main editor area displays the following code:

```
// Load Node Modules/Plugins
var gulp = require('gulp');
var concat = require('gulp-concat');
var uglify = require('gulp-uglify');
```

At the bottom of the editor, it says "Line 4, Column 37" and "Spaces: 4 JavaScript".

In this code, we have included gulp and two gulp plugins: `gulp-concat` and `gulp-uglify`. As you can now see, including a plugin into your gulpfile is quite easy. After we install each module or plugin using npm, you simply use node.js' `require()` function and pass it in the name of the module. You then assign it to a new variable so you can use it throughout your gulpfile.

This is node.js' way of handling modularity, and because a gulpfile is essentially a small node.js application, it adopts this practice as well.

Writing a task

All tasks in gulp share a common structure. Having reviewed the five methods at the beginning of this section, you will already be familiar with most of it. Some tasks might end up being larger than others, but they still follow the same pattern. To better illustrate how they work, let's examine a bare skeleton of a task. This skeleton is the basic "bone structure" of each task we will be creating. Studying this structure will make it incredibly simple to understand how parts of gulp work together to create a task. An example of a sample task is as follows:

```
gulp.task(name, function() {  
    return gulp.src(path)  
        .pipe(plugin)  
        .pipe(plugin)  
        .pipe(gulp.dest(path)) ;  
});
```

In the first line, we use the new `gulp` variable that we created a moment ago and access the `.task()` method. This creates a new task in our gulpfile. As you learned earlier, the task method accepts two arguments: a task name as a string and a callback function that will contain the actions we wish to run when this task is executed.

Inside the callback function, we refer to the `gulp` variable once more and then use the `.src()` method to provide the input to our task. As you learned earlier, the source method accepts a path or an array of paths to the files that we wish to process.

Next, we have a series of three `.pipe()` methods. In each of these pipe methods, we will specify which plugin we would like to use. This grouping of pipes is what we call our pipechain.

The data that we have provided gulp with in our source method will flow through our pipechain to be modified by each piped plugin that it passes through. The order of the pipe methods is entirely up to you. This gives you a great deal of control in how and when your data is modified.

You may have noticed that the final pipe is a bit different. At the end of our pipechain, we have to tell gulp to move our modified file somewhere. This is where the `.dest()` method comes into play. As we mentioned earlier, the destination method accepts a path that sets the destination of the processed file as it reaches the end of our pipechain. If `.src()` is our input, then `.dest()` is our output.

Reflection

To wrap up, take a moment to look at a finished gulpfile and reflect on the information that we just covered. This is the completed gulpfile that we will be creating from scratch in the next chapter, so don't worry if you still feel lost. This is just an opportunity to recognize the patterns and syntaxes that we have been studying so far.

In the next chapter, we will begin creating this file step by step:

```
// Load Node Modules/Plugins
var gulp = require('gulp');
var concat = require('gulp-concat');
var uglify = require('gulp-uglify');

// Process Styles
gulp.task('styles', function() {
    return gulp.src('css/*.css')
        .pipe(concat('all.css'))
        .pipe(gulp.dest('dist/'));
});

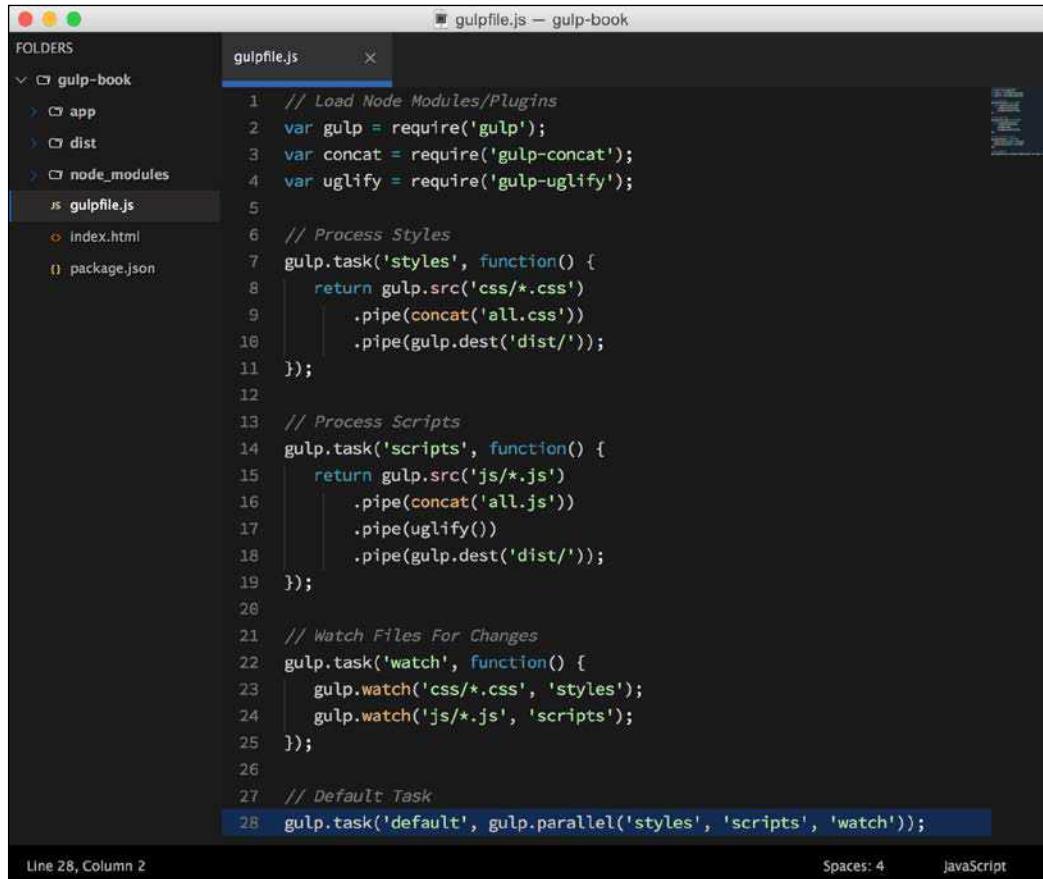
// Process Scripts
gulp.task('scripts', function() {
    return gulp.src('js/*.js')
        .pipe(concat('all.js'))
        .pipe(uglify())
        .pipe(gulp.dest('dist/'));
});

// Watch Files For Changes
gulp.task('watch', function() {
    gulp.watch('css/*.css', 'styles');
    gulp.watch('js/*.js', 'scripts');
});

// Default Task
gulp.task('default', gulp.parallel('styles', 'scripts', 'watch'));
```

Getting Started

The `gulpfile.js` file will look as follows:



A screenshot of a code editor window titled "gulpfile.js — gulp-book". The window shows a file tree on the left with a "FOLDERS" section containing "gulp-book", "app", "dist", and "node_modules". Inside "node_modules", there is a "gulpfile.js" file which is currently selected. The code editor displays the following JavaScript code:

```
// Load Node Modules/Plugins
var gulp = require('gulp');
var concat = require('gulp-concat');
var uglify = require('gulp-uglify');

// Process Styles
gulp.task('styles', function() {
  return gulp.src('css/*.css')
    .pipe(concat('all.css'))
    .pipe(gulp.dest('dist/'));
});

// Process Scripts
gulp.task('scripts', function() {
  return gulp.src('js/*.js')
    .pipe(concat('all.js'))
    .pipe(uglify())
    .pipe(gulp.dest('dist/'));
});

// Watch Files For Changes
gulp.task('watch', function() {
  gulp.watch('css/*.css', 'styles');
  gulp.watch('js/*.js', 'scripts');
});

// Default Task
gulp.task('default', gulp.parallel('styles', 'scripts', 'watch'));
```

Line 28, Column 2 Spaces: 4 JavaScript

Summary

In this chapter, we learned about how to use a command-line application to navigate a file system and create new files and folders. With this newfound knowledge, we scaffolded our project directory that we will use throughout the remainder of the book.

After completing our project structure, we installed node.js and learned the basics of how to use npm and understood how and why to install gulp both locally and globally.

To wrap up the chapter, we took a small glimpse into the anatomy of a gulpfile to prepare us for writing our own gulpfiles from scratch in the next chapter.

3

Performing Tasks with Gulp

We have spent a lot of time preparing you for this moment. It is finally time to write some code and improve your workflow!

For this simple project, we will use very basic "Hello, world!" type of examples to demonstrate how gulp accesses, modifies, and outputs our code and images.

Our project will create separate tasks that will process CSS, JavaScript, and images. For CSS, we will combine all of the files into a single file and then preprocess it to enable additional features in our code. For JavaScript, we will combine the files, check the code for errors, and minify it to reduce the overall file size. For images, we will use a plugin to compress them so that they can load more quickly and improve our project's performance.

Using gulp plugins

Without plugins, gulp is simply a means of connecting and organizing small bits of functionality. The plugins we are going to install will add the functionality we need to properly modify and optimize our code. Like gulp, all of the gulp plugins we will be using are installed via npm.

It is important to note that the gulp team cares deeply about their plugin ecosystem and spends a lot of time making sure they eliminate plugins that duplicate the functionality that has already been created. To enforce these plugin standards, they have implemented a blacklisting system that only shows the approved plugins. You can search for the approved plugins and modules by visiting <http://gulpjs.com/plugins>.



It is important to note that if you search for gulp plugins in the npm registry, you will be shown all the plugins, including the blacklisted ones. So, just to be safe, stick to the official plugin search results to weed out any plugins that might lead you down a wrong path.

Additionally, you can run gulp with the `--verify` flag to make it check whether any of your currently installed plugins and modules are blacklisted. In the following tasks, I will provide you with instructions on how to install gulp plugins as required. These will be installed in the same way we installed gulp in *Chapter 2, Getting Started*, except that here we will install multiple plugins at once. The code will look something like this:

```
npm install gulp-plugin1 gulp-plugin2 gulp-plugin3 --save-dev
```

This is simply a shorthand to save you time. You could just as easily run each of these commands separately, but it would only take more time:

```
npm install gulp-plugin1 --save-dev
npm install gulp-plugin2 --save-dev
npm install gulp-plugin3 --save-dev
```



Remember, on Mac and Linux systems, you may need to add in the additional sudo keyword to the beginning of your commands if you are in a protected area of your file system. Otherwise, you will receive permission errors and none of the modules will be installed.

The styles task

The first task we are going to add to our gulpfile will be our styles task. Our goal with this task is to combine all of our CSS files into a single file and then run those styles through a preprocessor such as **Sass**, **Less**, or **Myth**. In this example, we will use Myth, but you can simply substitute any other preprocessor that you would prefer to use.

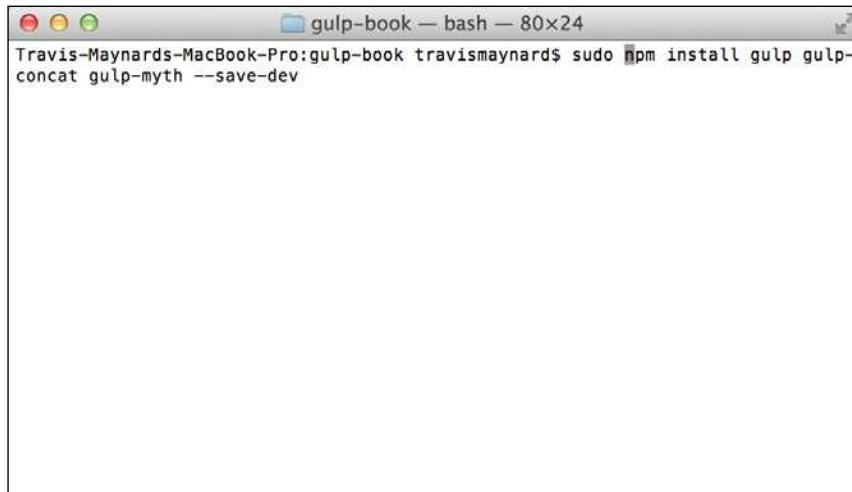
Installing gulp plugins

For this task, we will be using two plugins: `gulp-concat` and `gulp-myth`. As mentioned in the preceding section, we will install both of these tasks at the same time using the shortcut syntax. In addition to these plugins, we need to install gulp as well since this is the first task that we will be writing. For the remaining tasks, it won't be necessary to install gulp again, as it will already be installed locally in our project.

The command for installing gulp plugin is as follows:

```
npm install gulp gulp-concat gulp-myth --save-dev
```

The following two screenshots show the installation of the gulp plugin:



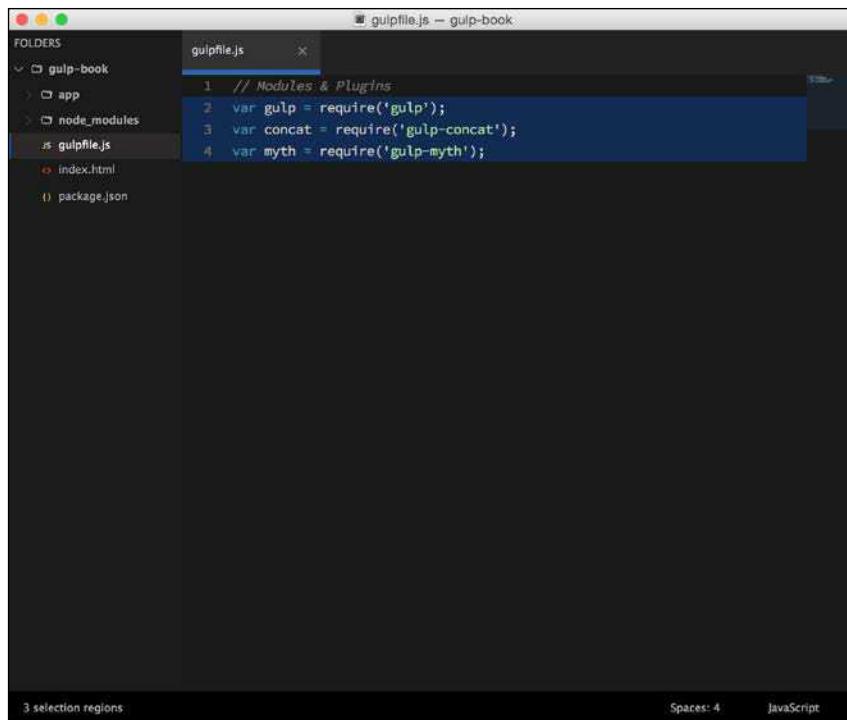
While running these commands, make sure that you're in the root directory of your project. If you're following the naming conventions used throughout this book, then the folder should be named `gulp-book`.

Including gulp plugins

Once complete, you will need to include references to those plugins at the beginning of your `gulpfile.js`. To do this, simply open `gulpfile.js` and add the following lines to it:

```
var gulp = require('gulp');
var concat = require('gulp-concat');
var myth = require('gulp-myth');
```

You can now match your `gulpfile` with the following screenshot:



```
// Modules & Plugins
var gulp = require('gulp');
var concat = require('gulp-concat');
var myth = require('gulp-myth');
```

Writing the styles task

With these references added, we can now begin to write our `styles` task. We will start off with the main task method and pass a string of `styles` to it as its identifier. This is the main method that will wrap all of the tasks we will be creating throughout the book. The code for the `task()` method is as follows:

```
gulp.task('styles', function() {
  // Code Goes Here
});
```

Next, you will need to tell gulp where it can find the source files that you wish to process. You instruct gulp by including a path to the file, but the path can contain globbing wildcards such as `*` to reference multiple files within a single directory. To demonstrate this, we will target all of the files that are inside of our `css` directory in our project.

```
gulp.task('styles', function() {
    return gulp.src('app/css/*.css')
        // Pipes Coming Soon
});
```

We have used the `*` globbing pattern to tell gulp that our source is every file with a `.css` extension inside of our `css` folder. This is a very valuable pattern that you will use throughout the writing of your tasks. Once our source has been set up, we can begin piping in our plugins to modify our data. We will begin by concatenating our source files into a single CSS file named `all.css`:

```
gulp.task('styles', function() {
    return gulp.src('app/css/*.css')
        .pipe(concat('all.css'))
        // More Pipes Coming Soon
});
```

In the preceding code, we added our `concat` reference that we included at the top of our `gulpfile` and passed it in a filename for the concatenated CSS file. In similar build systems, this would create a file and place it in a temporary location; however, with gulp, we can send this newly created file to the next step in our pipechain without writing out to any temporary files. Next, we will pipe in our concatenated CSS file into our preprocessor:

```
gulp.task('styles', function() {
    return gulp.src('app/css/*.css')
        .pipe(concat('all.css'))
        .pipe(myth())
});
```

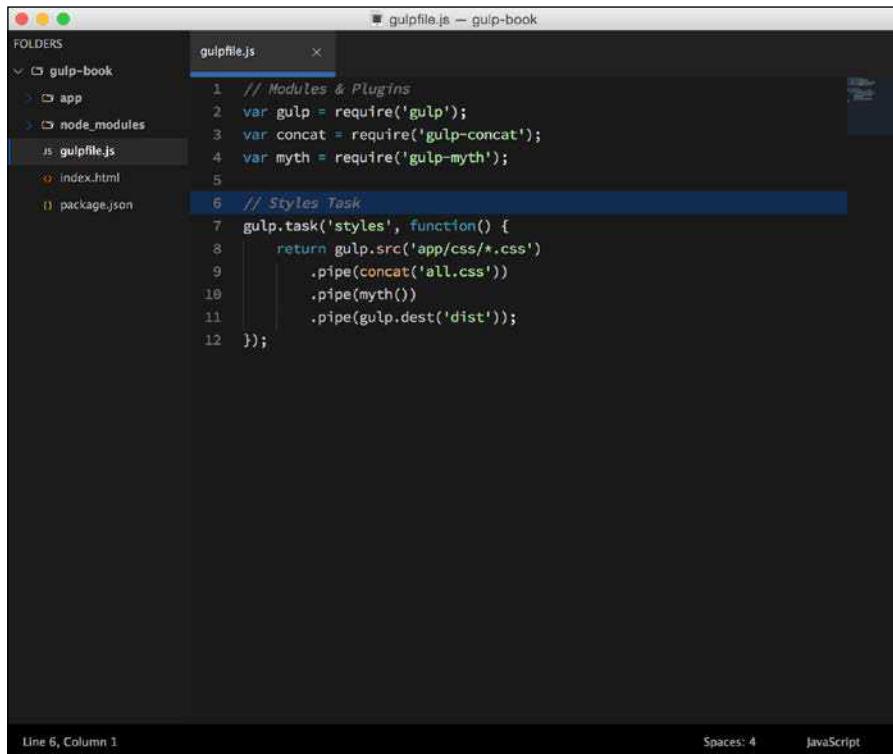
Finally, to finish the task, we must specify where we need to output our file. In our project, we will be outputting the file into a folder named `dist` that is located inside of our root project directory. To output a file, we will use gulp's `.dest()` method that we learned about in *Chapter 2, Getting Started*. This expects only a single argument, namely, the directory where you would like to output your processed file. The code for the `dest()` function is as follows:

```
gulp.task('styles', function() {
    return gulp.src('app/css/*.css')
```

Performing Tasks with Gulp

```
.pipe(concat('all.css'))  
.pipe(myth())  
.pipe(gulp.dest('dist'));  
});
```

You can now match your gulpfile with the following screenshot:



The screenshot shows a code editor window with a dark theme. On the left is a sidebar titled 'FOLDERS' showing a project structure: 'gulp-book' contains 'app', 'node_modules', and 'gulpfile.js'. The 'gulpfile.js' tab is active, displaying the following code:

```
// Modules & Plugins  
var gulp = require('gulp');  
var concat = require('gulp-concat');  
var myth = require('gulp-myth');  
  
// Styles Task  
gulp.task('styles', function() {  
  return gulp.src('app/css/*.css')  
    .pipe(concat('all.css'))  
    .pipe(myth())  
    .pipe(gulp.dest('dist'));  
});
```

At the bottom of the editor, it says 'Line 6, Column 1' and 'Spaces: 4 JavaScript'.

In the preceding code, we added our final pipe with the `.dest()` method and supplied it with our `dist` directory that I mentioned in one of the previous sections. This task will now put our concatenated and preprocessed file into our `dist` directory for us to include it in our application. This task is now essentially complete! We will continue to add additional functionality to it as we progress through the book, but for now our core functionality is in place.

Other preprocessors

It is important to note that concatenating our files is often not really necessary when using a preprocessor such as Sass. This is because it already includes an `@import` feature that allows you to separate your CSS files into partials based on their specific purpose and then pulls them all into a single file.

If you are using this functionality within Sass, then we can very easily modify our task by installing the `gulp-sass` plugin and rearranging our pipes. To do so, you would simply install the `gulp-sass` plugin and then modify your task as follows:

```
npm install gulp-sass --save-dev
```

The code for `gulp-sass` task is as follows:

```
gulp.task('styles', function() {
    return gulp.src('app/css/*.scss')
        .pipe(sass())
        .pipe(gulp.dest('dist'));
});
```

You can now remove the concatenation pipe as the `gulp-sass` plugin will hit those imports and pull everything up together for you. So, in this case, all you would need to do is simply change the source files over to `.scss` and remove the initial pipe that we used to concatenate our files. After those changes have been made, the pipechain will continue to work as expected.

Reviewing the styles task

Our `styles` task will first take in our CSS source files and then concatenate them into a single file that we have called `all.css`. Once they have been concatenated, we are going to pass our new `all.css` file into our pipe that will then preprocess it using Myth (again, you can substitute any preprocessor you prefer to use). Finally, we will save that concatenated and preprocessed file in our `dist` directory where we can finally include it in our website or application.

The scripts task

The second task will be to handle all of the JavaScript files in the project. The goal with this task is to concatenate the code into a single file, minify that code into a smaller file size, and check the code for any errors that may prevent it from running properly.

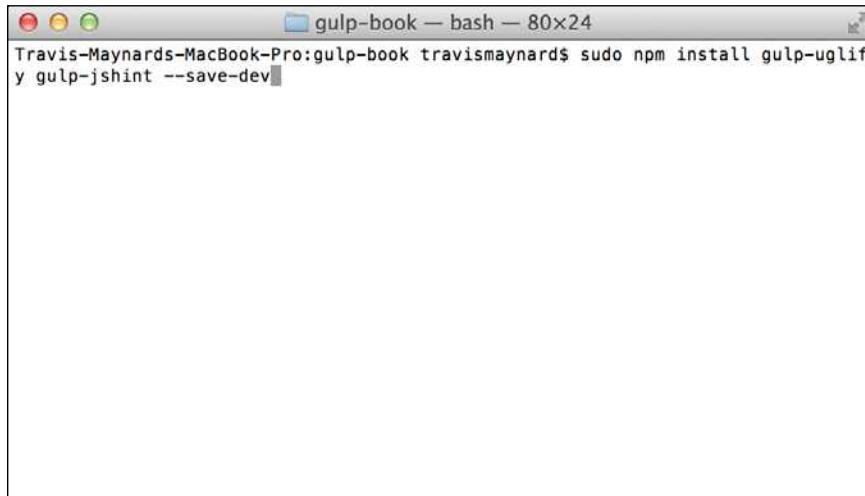
Installing gulp plugins

For this task, we will use three plugins: `gulp-concat`, `gulp-uglify`, and `gulp-jshint` to accomplish our goals. Like before, we will install these tasks using the shortcut syntax to save time. Since we installed `gulp` and `gulp-concat` locally while we were writing the `styles` task, it is not necessary to install them again. The command for installing `uglify` and `jshint` is as follows:

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

Performing Tasks with Gulp

The next two screenshots gives you a picture of what this will look like:



```
Travis-Maynards-MacBook-Pro:gulp-book travismaynard$ sudo npm install gulp-uglify  
y gulp-jshint --save-dev
```



```
Travis-Maynards-MacBook-Pro:gulp-book travismaynard$ sudo npm install gulp-uglify  
y gulp-jshint --save-dev  
npm WARN package.json gulp-book@1.0.0 No repository field.  
npm WARN package.json gulp-book@1.0.0 No README data  
gulp-uglify@1.0.1 node_modules/gulp-uglify  
├── deepmerge@0.2.7  
├── through2@0.6.2 (xtend@4.0.0, readable-stream@1.0.32)  
├── uglify-js@2.4.15 (uglify-to-browserify@1.0.2, async@0.2.10, optimist@0.3.7,  
source-map@0.1.34)  
└── vinyl-sourcemaps-apply@0.1.4 (source-map@0.1.40)  
    ├── gulp-util@3.0.1 (lodash._reinterpolate@2.4.1, dateformat@1.0.8, minimist@1.1  
.0, chalk@0.5.1, multipipe@0.1.1, vinyl@0.4.3, lodash.template@2.4.1, lodash@2.4  
.1)  
  
gulp-jshint@1.8.5 node_modules/gulp-jshint  
├── rcooader@0.1.2 (rcfinder@0.1.8)  
├── through2@0.6.2 (xtend@4.0.0, readable-stream@1.0.32)  
├── minimatch@1.0.0 (sigmund@1.0.0, lru-cache@2.5.0)  
├── gulp-util@3.0.1 (lodash._reinterpolate@2.4.1, dateformat@1.0.8, minimist@1.1  
.0, chalk@0.5.1, multipipe@0.1.1, lodash.template@2.4.1, vinyl@0.4.3)  
└── lodash@2.4.1  
    └── jshint@2.5.6 (strip-json-comments@1.0.1, underscore@1.6.0, exit@0.1.2, shell  
js@0.3.0, console-browserify@1.1.0, cli@0.6.4, htmlparser2@3.7.3)  
Travis-Maynards-MacBook-Pro:gulp-book travismaynard$
```



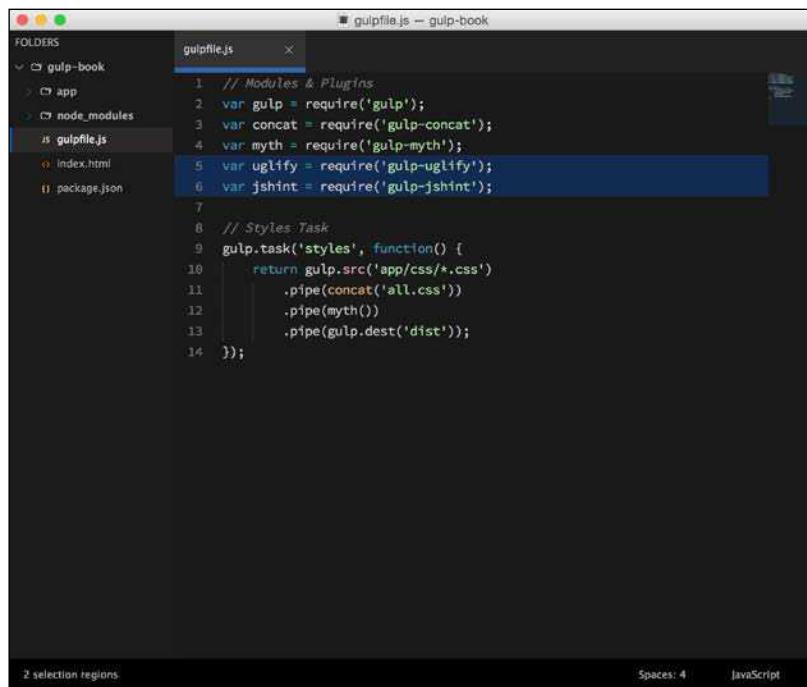
When running these commands, make sure that you're in the root directory of your project. If you're following the naming conventions used throughout this book, then the folder should be named `gulp-book`. Additionally, you may need to precede this command with `sudo` if you are running into permission errors.

Including gulp plugins

At the top of our `gulpfile`, we need to add in the new gulp plugins that we have installed for this task. Once added, the top of your `gulpfile` should look like this:

```
var gulp = require('gulp');
var concat = require('gulp-concat');
var myth = require('gulp-myth');
var uglify = require('gulp-uglify'); // Newly Added
var jshint = require('gulp-jshint'); // Newly Added
```

The `gulpfile.js` file will look as follows:



```
// Modules & Plugins
var gulp = require('gulp');
var concat = require('gulp-concat');
var myth = require('gulp-myth');
var uglify = require('gulp-uglify');
var jshint = require('gulp-jshint');

// Styles Task
gulp.task('styles', function() {
  return gulp.src('app/css/*.css')
    .pipe(concat('all.css'))
    .pipe(myth())
    .pipe(gulp.dest('dist'));
});
```

Writing the scripts task

Once the references to the new gulp plugins have been added, we can begin writing the `scripts` task. Like with the `styles` task, we will begin by writing out the task wrapper; only this time, we will provide it with the string, `'scripts'`, as the name for the task:

```
gulp.task('scripts', function() {
  // Code Goes Here
});
```

Like before, we will need to tell gulp where the JavaScript source files are located using the `.src` method. We will be using the same `*` globbing pattern that we used before to target all of the JavaScript source files in our project directory:

```
gulp.task('scripts', function() {
    return gulp.src('app/js/*.js')
        // Pipes Coming Soon
});
```

Next, we can begin adding in the pipes and plugins that we will be using in this task. The first plugin that we will use for the scripts task is `gulp-jshint`. JSHint is a tool that is used to analyze JavaScript files and report any errors that could potentially break an application. The JSHint plugins also require an additional pipe that it can use to report when errors take place. For this example, we are going to use the default reporter. If any errors are found, they will be output into the command-line application from where gulp is being run:

```
gulp.task('scripts', function() {
    return gulp.src('app/js/*.js')
        .pipe(jshint())
        .pipe(jshint.reporter('default'))
        // More Pipes Coming Soon
});
```

The next pipe we will use is `gulp-concat`. As you might recall, this is the same plugin we started with while building the styles task. As with CSS, we will concatenate all of the JavaScript files into a single file to reduce the number of requests that are needed to load our website:

```
gulp.task('scripts', function() {
    return gulp.src('app/js/*.js')
        .pipe(jshint())
        .pipe(jshint.reporter('default'))
        .pipe(concat('all.js'))
        // More Pipes Coming Soon
});
```

The next plugin we will use is `gulp-uglify`, which will minify the code to reduce the file size of the concatenated JavaScript file. This is also an important and very valuable optimization:

```
gulp.task('scripts', function() {
    return gulp.src('app/js/*.js')
        .pipe(jshint())
```

```

    .pipe(jshint.reporter('default'))
        .pipe(concat('all.js'))
    .pipe(uglify())
        // Another Pipe Coming Soon
);
}

```

Finally, like the styles task, the final pipe will use gulp's built-in `.dest` method to place the processed file in the project's `dist` directory:

```

gulp.task('scripts', function() {
    return gulp.src('app/js/*.js')
        .pipe(jshint())
        .pipe(jshint.reporter('default'))
    .pipe(concat('all.js'))
        .pipe(uglify())
        .pipe(gulp.dest('dist'));
});

```

The following screenshot is what your `gulpfile.js` should look like after the scripts task has been added:

```

// Modules & Plugins
var gulp = require('gulp');
var concat = require('gulp-concat');
var myth = require('gulp-myth');
var uglify = require('gulp-uglify');
var jshint = require('gulp-jshint');

// Styles Task
gulp.task('styles', function() {
    return gulp.src('app/css/*.css')
        .pipe(concat('all.css'))
        .pipe(myth())
        .pipe(gulp.dest('dist'));
});

// Scripts Task
gulp.task('scripts', function() {
    return gulp.src('app/js/*.js')
        .pipe(jshint())
        .pipe(jshint.reporter('default'))
        .pipe(concat('all.js'))
        .pipe(uglify())
        .pipe(gulp.dest('dist'));
});

```

Line 16, Column 1 Spaces: 4 JavaScript

Reviewing the scripts task

The scripts task will take in all of the JavaScript files in the `js` directory inside the project and then check each file for errors using the JSHint plugin. If any errors are found, they would be displayed in the command-line application using the default JSHint reporter. Next, our JavaScript files will be concatenated into a single `all.js` file and handed off to the minification plugin, UglifyJS, to reduce the overall size of the file. Finally, we will output the concatenated and minified file to the project's `dist` directory.

After wrapping up the second task, you should now be able to recognize the patterns that we discussed in *Chapter 2, Getting Started*. Understanding these patterns and structures is really the most important part of learning gulp, as everything else you will continue to learn simply builds on top of them.

The images task

The third task will handle all of the image processing. This task will be a bit smaller than the first two as it will only use a single plugin. The goal for this task is to optimize our images by minifying them, which will help reduce the load times of the project.

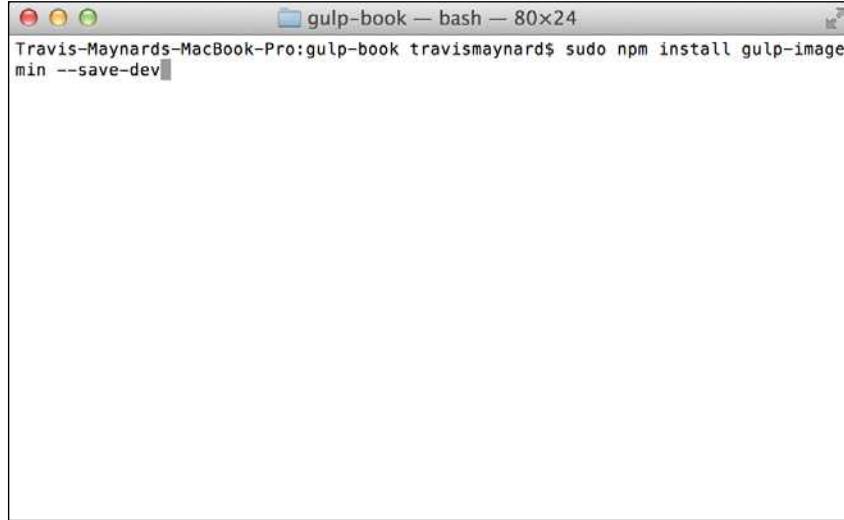
Installing gulp plugins

To install gulp plugins, we will use only one plugin: `gulp-imagemin`. Like each task before it, you will need to install this locally and save it to your development dependencies.

The command for installing the `imagemin` plugin is as follows:

```
npm install gulp-imagemin --save-dev
```

The following two screenshots shows the installation of the imagemin plugin:



```
gulp-book — bash — 80x24
Travis-Maynards-MacBook-Pro:gulp-book travismaynard$ sudo npm install gulp-image
min --save-dev
```



```
gulp-book — bash — 80x24
Travis-Maynards-MacBook-Pro:gulp-book travismaynard$ sudo npm install gulp-image
min --save-dev
npm WARN package.json gulp-book@1.0.0 No repository field.
npm WARN package.json gulp-book@1.0.0 No README data
/
> gifsicle@1.0.3 postinstall /Users/travismaynard/.Trash/gulp-book/node_modules/
  gulp-imagemin/node_modules/imagemin/node_modules/imagemin-gifsicle/node_modules/
  gifsicle
> node lib/install.js

  downloading : https://raw.github.com/imagemin/gifsicle-bin/v1.0.3/vendor/osx/g
  ifsicle
    progress : [=====] 100% 0.0s
✓ pre-build test passed successfully!

> pngquant-bin@1.0.1 postinstall /Users/travismaynard/.Trash/gulp-book/node_modu
  les/gulp-imagemin/node_modules/imagemin/node_modules/imagemin-pngquant/node_modu
  les/pngquant-bin
> node lib/install.js

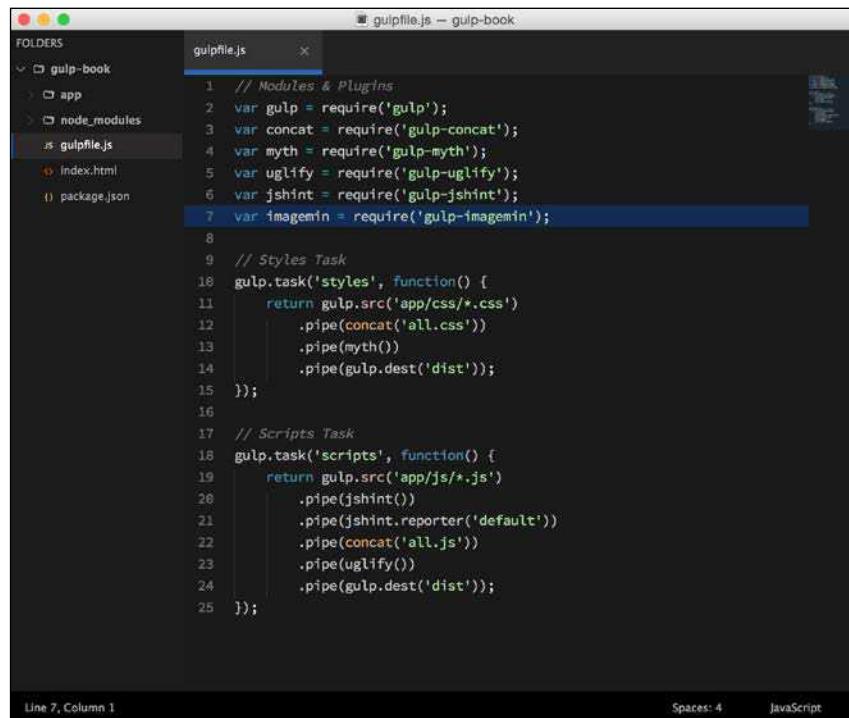
  downloading : https://raw.github.com/imagemin/pngquant-bin/v1.0.1/vendor/osx/p
  ngquant
    progress : [=====] 100% 0.0s
✓ pre-build test passed successfully!
```

[ When running these commands, make sure that you're in the root directory of your project. If you're following the naming conventions used throughout this book, then the folder should be named `gulp-book`. Additionally, you may need to precede this command with `sudo` if you are running into permission errors.]

Including gulp plugins

With the new plugin installed, we can now add it to the top of the `gulpfile` along with the other code:

```
var gulp = require('gulp');
var concat = require('gulp-concat');
var myth = require('gulp-myth');
var uglify = require('gulp-uglify');
var jshint = require('gulp-jshint');
var imagemin = require('gulp-imagemin'); // Newly Added
```



The screenshot shows a code editor window with the file `gulpfile.js` open. The file is part of a project structure with folders `gulp-book`, `app`, `node_modules`, and `index.html`. The `package.json` file is also present. The code in `gulpfile.js` defines two tasks: `styles` and `scripts`. The `styles` task concatenates CSS files, runs them through the `myth` plugin, and then pipes the result to the `dest` plugin. The `scripts` task concatenates JavaScript files, runs them through `jshint`, generates a reporter, concatenates them into `all.js`, runs them through `uglify`, and then pipes the result to the `dest` plugin. The line `var imagemin = require('gulp-imagemin');` is highlighted with a blue selection bar.

Writing the images task

As you're probably now used to, we will start off the images task by including the main task wrapper and then using gulp's `.src` method to target all of the images in the project:

```
gulp.task('images', function() {
  return gulp.src('app/img/*')
    // Pipes Coming Soon
});
```

The only difference here is that we have not specified a file name; we are including every file that resides in the `img` folder. The reason for this is that throughout the course of development, it is likely that the project will use more than a single image file type. We already know that there will only be images in that directory, so by targeting all of the files, we are proactively bypassing future limitations for this task.

All that is left is to pipe in the newly added plugin followed by gulp's .dest method, which will save the optimized images into a new img folder alongside the optimized CSS and JavaScript files:

```
gulp.task('images', function() {
  return gulp.src('app/img/*')
    .pipe(imagemin())
    .pipe(gulp.dest('dist/img')));
});
```

The following screenshot shows the image task code:

The screenshot shows a code editor with a sidebar titled "FOLDERS" containing "gulp-book", "app", "node_modules", "gulpfile.js", "Index.html", and "package.json". The main area has a title bar "gulpfile.js — gulp-book". The code in "gulpfile.js" is as follows:

```
var uglify = require('gulp-uglify');
var jshint = require('gulp-jshint');
var imagemin = require('gulp-imagemin');

// Styles Task
gulp.task('styles', function() {
  return gulp.src('app/css/*.css')
    .pipe(concat('all.css'))
    .pipe(myth())
    .pipe(gulp.dest('dist'));
});

// Scripts Task
gulp.task('scripts', function() {
  return gulp.src('app/js/*.js')
    .pipe(jshint())
    .pipe(jshint.reporter('default'))
    .pipe(concat('all.js'))
    .pipe(uglify())
    .pipe(gulp.dest('dist'));
});

// Images Task
gulp.task('images', function() {
  return gulp.src('app/img/*')
    .pipe(imagemin())
    .pipe(gulp.dest('dist/img'));
});
```

Reviewing the images task

The images task is the smallest task yet and performs only a single action on our data. It first supplies gulp with all of the images in our project and then runs them through our `imagemin` plugin to minify each file and reduce their file sizes. After this is complete, the optimized files are then passed into gulp's `.dest` method where they are saved inside a new `img` folder, inside of the project's `dist` directory.

The watch task

So far, all of the tasks that we have written are actually only capable of running once. Once they complete, their job is considered done. However, this isn't very helpful as we would end up having to go back to our command line to run them again every time we make a change to our files. This is where gulp's `.watch` method comes into play. The `watch` method's job is to specifically look for changes to files and then respond by running tasks in relation to those changes. Since we will be focusing on CSS, JavaScript, and images independently, we will need to specify three separate watch methods as well. To better organize these watch methods, we will create an additional watch task that will serve as a reference and an easy way to organize them within our `gulpfile`.

Writing the watch task

Since the `watch` method is built into gulp as a core feature, no new plugins are needed. So we can move straight to actually writing the task itself. Additionally, we will not use the `.src` or `.dest` methods in this task as they have already been specified in the previous tasks.

As always, the first step in creating a task is to write out the main task method and provide it with a name. This task will be named `watch`, which is shown in the following code:

```
gulp.task('watch', function() {  
    // Code Goes Here  
});
```

If you can recall from *Chapter 2, Getting Started*, the `.watch` method accepts two arguments. The first is the path to the files that we want to monitor for changes, and the second is the name of the task that you wish to run if any of those files are changed. If you wish to run multiple tasks upon changes to your watched files, you can use the `.parallel` and `.series` tasks that we mentioned in *Chapter 2, Getting Started*.

These methods will allow you to run the tasks together at the same time or in the order that they are specified.

For our example, this wont be necessary as we only need to run a single task for each set of watched files.

With this in mind, we can now create each of our watch methods.



The first argument will usually match the path that we provide to our `.src` method within the previous tasks we have written.



To watch and run a single task upon a change, you will use the following syntax:

```
gulp.task('watch', function() {
    gulp.watch('app/css/*.css', 'styles');
    gulp.watch('app/js/*.js', 'scripts');
    gulp.watch('app/img/*', 'images');
});
```

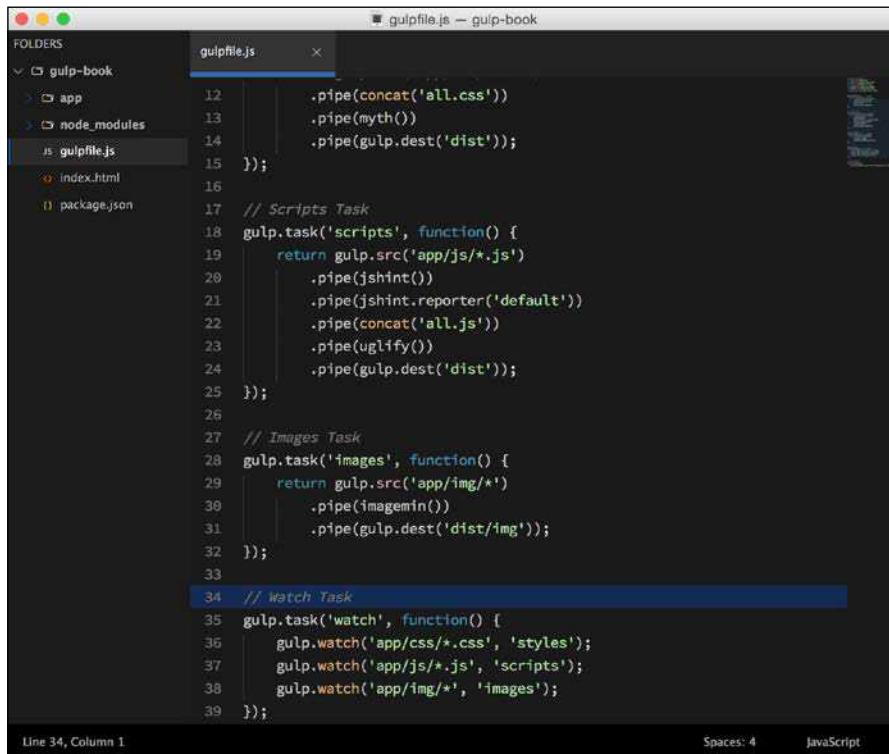
If your project requires you to run multiple tasks upon a change, you will use one of the following syntaxes:

- `gulp.watch('app/css/*.css', gulp.parallel('firstTask', 'secondTask'));`
- `gulp.watch('app/js/*.js', gulp.series('thirdTask', 'fourthTask'));`

Simply replace the task string with the method that you need to use for your project and then pass in all tasks that need to be ran as arguments.

Performing Tasks with Gulp

The method that you use will depend on your project. If any of your tasks require another task to finish completely before moving to the next, then you must use the series method. The series method will tell gulp that each task must execute in the order that they are specified. This gives you far more control over how you execute tasks and structure your gulpfile.



A screenshot of a code editor showing a Gulpfile.js file. The file structure on the left shows a folder named 'gulp-book' containing 'app', 'node_modules', and 'gulpfile.js'. The 'gulpfile.js' tab is selected. The code in the editor is as follows:

```
12     .pipe(concat('all.css'))
13     .pipe(myth())
14     .pipe(gulp.dest('dist'));
15 });
16
17 // Scripts Task
18 gulp.task('scripts', function() {
19     return gulp.src('app/js/*.js')
20         .pipe(jshint())
21         .pipe(jshint.reporter('default'))
22         .pipe(concat('all.js'))
23         .pipe(uglify())
24         .pipe(gulp.dest('dist'));
25 });
26
27 // Images Task
28 gulp.task('images', function() {
29     return gulp.src('app/img/*')
30         .pipe(imagemin())
31         .pipe(gulp.dest('dist/img'));
32 });
33
34 // Watch Task
35 gulp.task('watch', function() {
36     gulp.watch('app/css/*.css', 'styles');
37     gulp.watch('app/js/*.js', 'scripts');
38     gulp.watch('app/img/*', 'images');
39 });

Line 34, Column 1
Spaces: 4
JavaScript
```

In the preceding code, we created three `.watch` methods: one for our CSS, one for our JavaScript, and one for our images. In each of these, we provide the method with the path to those files and follow that up with the name of the task we want to run if those files are modified. When this task is run, gulp will continuously look for changes to the files located in those paths, and if any of those files change, gulp will run the task that is specified in the second argument. This will continue to happen until gulp is manually stopped.

Reviewing the watch task

Watch task is one of the most unique tasks that we have written simply because it is more of a "helper task" that provides helpful functionality. It's not using or running any build-specific actions; it is only giving us more control over when our tasks can be run.

The default task

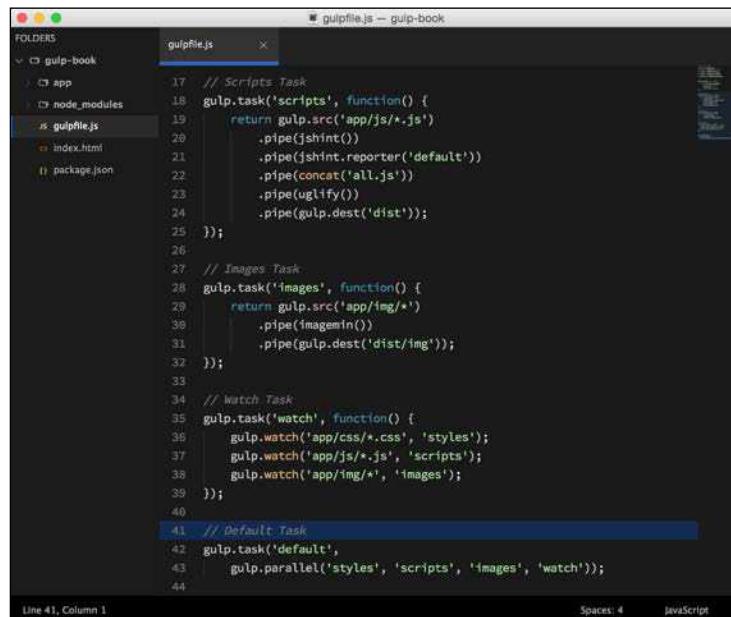
Our final task is the `default` task, and it is best considered as the entry point for our `gulpfile`. The purpose of this task is to gather and execute any tasks that `gulp` needs to run by default.

Writing the default task

The `default` task is the smallest and the most simple task in our `gulpfile` and will only take up a single line of code. For this task, we only need to provide it with the name `default` and the name of a single task or the parallel or series methods containing the multiple tasks that we wish to run.

```
gulp.task('default', gulp.parallel('styles', 'scripts', 'images',
  'watch'));
```

After adding the default task, our `gulpfile` should look like this:



```

FOLDERS
  - gulp-book
    - app
    - node_modules
    - gulpfile.js
    - index.html
  package.json

  gulpfile.js
  17 // Scripts Task
  18 gulp.task('scripts', function() {
  19   return gulp.src('app/js/*.js')
  20     .pipe(jshint())
  21     .pipe(jshint.reporter('default'))
  22     .pipe(concat('all.js'))
  23     .pipe(uglify())
  24     .pipe(gulp.dest('dist'));
  25 });
  26
  27 // Images Task
  28 gulp.task('images', function() {
  29   return gulp.src('app/img/*')
  30     .pipe(imagemin())
  31     .pipe(gulp.dest('dist/img'));
  32 });
  33
  34 // Watch Task
  35 gulp.task('watch', function() {
  36   gulp.watch('app/css/*.css', 'styles');
  37   gulp.watch('app/js/*.js', 'scripts');
  38   gulp.watch('app/img/*', 'images');
  39 });
  40
  41 // Default Task
  42 gulp.task('default',
  43   gulp.parallel('styles', 'scripts', 'images', 'watch'));
  44
  
```

Line 41, Column 1 Spaces: 4 JavaScript

This code will run each of our tasks once, including our watch task. The watch task will continuously check for changes to our files after the initial round of processing is complete and re-run each of our tasks when the related files change.

Completed gulpfile

Congratulations! You have written your very first `gulpfile`. However, this is only the beginning. We have a lot more to add in the coming chapters, including using `node.js` modules for advanced tasks as well as some great tips and improvements. Before we move on, let's take a moment to review the completed file that we created in this chapter. This is a great opportunity to compare your file with the code given and ensure that your code matches with what we have written so far. This may save your debugging time later!

The completed `gulpfile` would be as follows:

```
// Modules & Plugins
var gulp = require('gulp');
var concat = require('gulp-concat');
var myth = require('gulp-myth');
var uglify = require('gulp-uglify');
var jshint = require('gulp-jshint');
var imagemin = require('gulp-imagemin');

// Styles Task
gulp.task('styles', function() {
    gulp.src('app/css/*.css')
        .pipe(concat('all.css'))
        .pipe(myth())
        .pipe(gulp.dest('dist'));
});

// Scripts Task
gulp.task('scripts', function() {
    gulp.src('app/js/*.js')
        .pipe(jshint())
        .pipe(jshint.reporter('default'))
        .pipe(concat('all.js'))
        .pipe(uglify())
        .pipe(gulp.dest('dist'));
});

// Images Task
gulp.task('images', function() {
```

```
gulp.src('app/img/*')
    .pipe(imagemin())
    .pipe(gulp.dest('dist/img'));
});

// Watch Task
gulp.task('watch', function() {
    gulp.watch('app/css/*.css', 'styles');
    gulp.watch('app/js/*.js', 'scripts');
    gulp.watch('app/img/*', 'images');
});

// Default Task
gulp.task('default', gulp.parallel('styles', 'scripts', 'images',
'watch'));
```

Running tasks

Now that our `gulpfile` is complete, it is now time to learn how to run each of the tasks that we have written. In this section, we will learn more about our `default` task and how to target and run any of our tasks independently.

Running the default task

In most cases, `gulpfiles` are created to be ran with a single one word command, `gulp`. Upon running this command, `gulp` will run the task with the name `default` in our `gulpfile`. As you may recall from the previous section, that is why it is considered to be the entry point. When running `gulp` like this, without any parameters, it is built to always run the default task that can be used to run any number of tasks that we wish to include.

Running a single task

Some projects may require that a task be run independently and manually as a certain step in the workflow process. If you need to run any of the tasks manually, you can do so by simply separating your `gulp` command with a single space and then listing the name of the task that you wish to run. For example, the following command will only run our `styles` task:

```
gulp styles
```

You can do this with any of the tasks that you have included in your `gulpfile`, even your `watch` or `default` tasks. The important thing to remember is that if you don't specify a task, then `gulp` will automatically choose to run the default task for you.

So, consider that you run the following code:

```
gulp default
```

This is the same as running the following:

```
gulp
```

Stopping a watch task

Tasks are designed to run through their process and once they are completed, gulp will exit and return you to your command prompt. However, using the `.watch` method instructs gulp to continue listening for changes to your files beyond the initial execution. So, once a task is executed that uses a `.watch` method, gulp will not stop unless it runs into an error or until you specifically instruct it to stop.

For beginners, this can be quite confusing especially if you are still getting accustomed to your command line. The important thing to remember is that you must always manually stop an operation in progress by using the `Ctrl + C` key combination. This will end the processing application and return you to your command prompt to run more commands.

Summary

In this chapter, we learned how to write and run a fully functional `gulpfile` from the ground up. In it, we created three focused tasks to process our CSS, JavaScript, and image files.

Our CSS task joins together all of our CSS files and then passes the joined file through a preprocessor so that we can use cutting-edge CSS features, such as variables and mathematical calculations.

Our JavaScript task uses a linter to analyze our code for any errors, concatenate those files, and minify the combined file to reduce its overall file size. Finally, we created tasks to optimize our image files, watch our source files for changes, and determine which tasks are run by default.

In the next chapter, we will take a look at how to use regular node.js modules in place of gulp plugins to add advanced functionality to your projects.

4

Using Node.js Modules for Advanced Tasks

Using gulp plugins is the easiest way to add functionality to your workflow. However, the actions that you need to perform inside your tasks are sometimes better off being written using plain node.js modules.

In this chapter, we will cover the following topics:

- Common usage of plain node.js modules
- When and why these modules should be used in place of gulp plugins
- Static server, BrowserSync, and Browserify

Why use plain node.js modules?

A common misunderstanding and topic of confusion for gulp beginners is when and why to use plain node.js modules in place of using or creating a new gulp plugin. Generally, the best practice is that if you can use plain node.js modules, then you should use plain node.js modules.

Gulp was built on the Unix philosophy that we can pull together many smaller, single-purpose applications to perform more complex actions. With this philosophy we are never duplicating work or creating redundant code. Additionally, it is easier to test the expectations of each smaller application than it would be to test a large collection of duplicated code.

The gulp team spends a lot of time ensuring that their plugin ecosystem maintains the highest quality. Part of ensuring this is making sure that no gulp plugin deviates from this core philosophy. Any gulp plugins that don't follow it are blacklisted and will not be shown to other users in the official gulp plugin search. This is very important to remember when looking for plugins to use, and if you ever plan on creating a plugin yourself. Don't duplicate work that has already been done. If you would like to help improve a plugin, contact the maintainer and work together to improve it for everyone!

If we all decided to create our own version of every plugin, then the ecosystem would be inundated with duplication, which would only confuse users and damage the overall perception of gulp as a tool.



If you're ever unsure if the plugins you are using have been blacklisted, you can run the `gulp --verify` command to check if they are included on the official gulp blacklist.



Static server

For quick and easy distribution, having the ability to spin up a small file server can be a great time saver and will prevent the need to run larger server software such as Apache or Nginx.

For this task, instead of using a gulp plugin we are going to use the Connect middleware framework module. Middleware is a small layer that allows us to build additional functionality into our applications, or in this case our gulp tasks.

Connect itself only acts as the framework to pull in additional functionality, so in addition to Connect we will need to install the plugin that we wish to use. To spin up a static server, we will be using the `serve-static` node.js module.

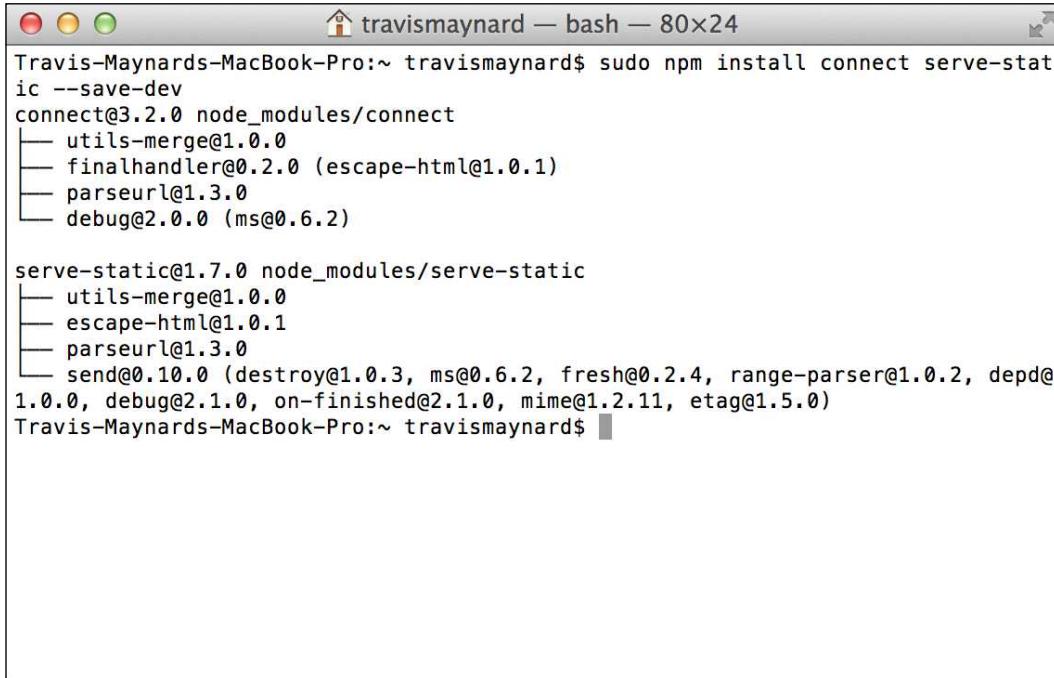
Installing modules

Installing plain node.js modules is exactly the same process as installing gulp plugins because, despite the gulp focus, gulp plugins are still node.js modules at heart. The modules we will be using for this specific task are `connect` and `serve-static`.

To install `connect` and `serve-static`, we will run the following command:

```
npm install connect serve-static --save-dev
```

After running this command, your output should look something like the following:



The screenshot shows a terminal window titled "travismaynard — bash — 80x24". The command run was "sudo npm install connect serve-static --save-dev". The output lists the dependencies for "connect" and "serve-static".

```
Travis-Maynards-MacBook-Pro:~ travismaynard$ sudo npm install connect serve-static
ic --save-dev
connect@3.2.0 node_modules/connect
└── utils-merge@1.0.0
    ├── finalhandler@0.2.0 (escape-html@1.0.1)
    ├── parseurl@1.3.0
    └── debug@2.0.0 (ms@0.6.2)

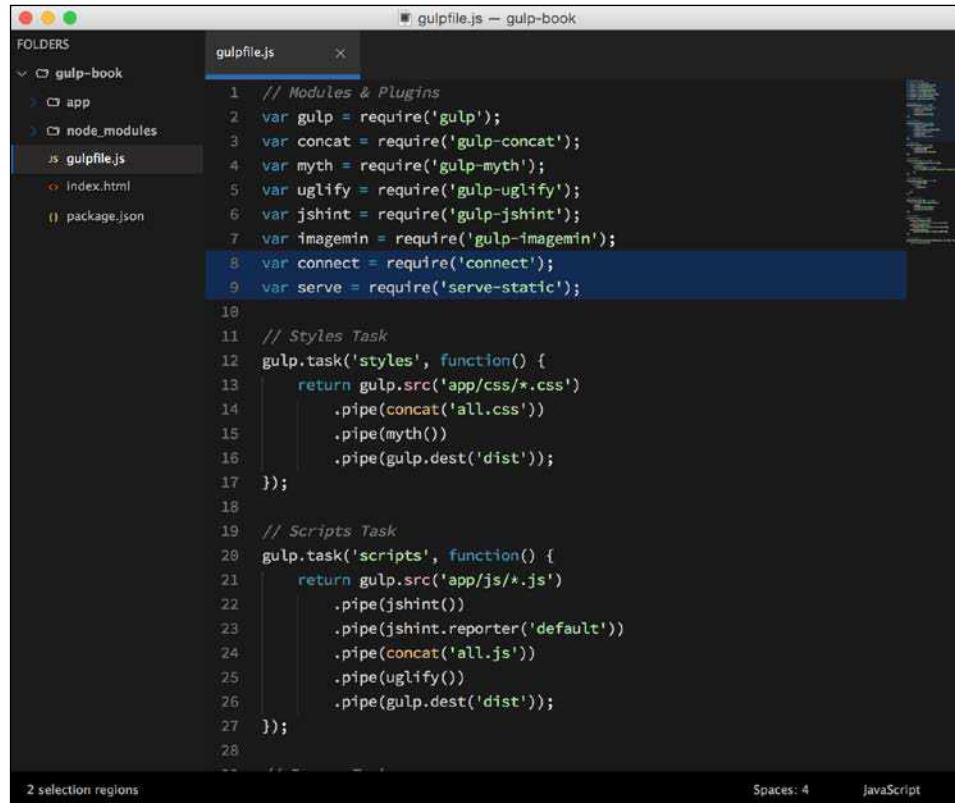
serve-static@1.7.0 node_modules/serve-static
└── utils-merge@1.0.0
    ├── escape-html@1.0.1
    ├── parseurl@1.3.0
    └── send@0.10.0 (destroy@1.0.3, ms@0.6.2, fresh@0.2.4, range-parser@1.0.2, depd@1.0.0, debug@2.1.0, on-finished@2.1.0, mime@1.2.11, etag@1.5.0)
Travis-Maynards-MacBook-Pro:~ travismaynard$
```

Including modules

As you might expect, we will include any plain node.js modules in the same way that we included our gulp plugins from the previous chapter. We will be adding these two to the bottom of our code.

```
// Modules & Plugins
var gulp = require('gulp');
var concat = require('gulp-concat');
var myth = require('gulp-myth');
var uglify = require('gulp-uglify');
var jshint = require('gulp-jshint');
var imagemin = require('gulp-imagemin');
var connect = require('connect'); // Added
var serve = require('serve-static'); // Added
```

The updated gulpfile should look like this:



A screenshot of a code editor window titled "gulpfile.js — gulp-book". The file contains the following code:

```
// Modules & Plugins
var gulp = require('gulp');
var concat = require('gulp-concat');
var myth = require('gulp-myth');
var uglify = require('gulp-uglify');
var jshint = require('gulp-jshint');
var imagemin = require('gulp-imagemin');
var connect = require('connect');
var serve = require('serve-static');

// Styles Task
gulp.task('styles', function() {
  return gulp.src('app/css/*.css')
    .pipe(concat('all.css'))
    .pipe(myth())
    .pipe(gulp.dest('dist'));
});

// Scripts Task
gulp.task('scripts', function() {
  return gulp.src('app/js/*.js')
    .pipe(jshint())
    .pipe(jshint.reporter('default'))
    .pipe(concat('all.js'))
    .pipe(uglify())
    .pipe(gulp.dest('dist'));
});
```

The code editor shows syntax highlighting for JavaScript. There are two selection regions indicated by blue highlights on lines 8 and 27. The status bar at the bottom right shows "Spaces: 4" and "JavaScript".

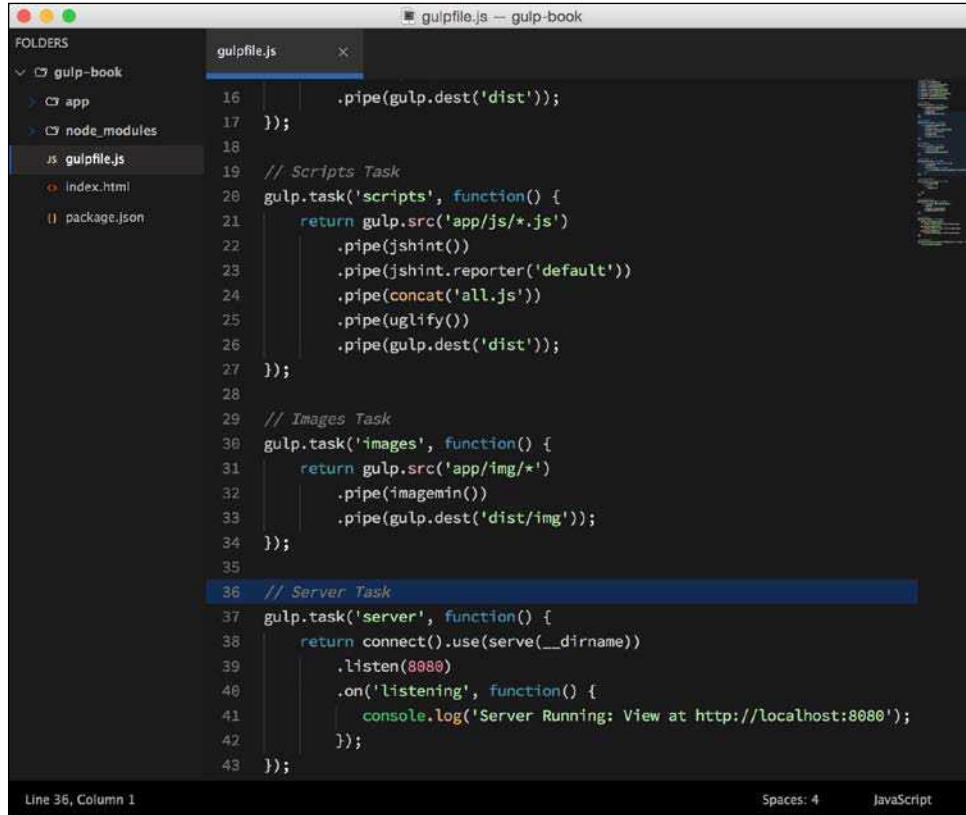
Writing a server task

Once our node.js modules have been installed and included, we can begin writing our new task. We will introduce some more advanced node.js-specific syntax, but it will most likely feel somewhat familiar to the tasks we created in the previous chapter.

Our server task will look like this:

```
gulp.task('server', function() {
  return connect().use(serve(__dirname))
    .listen(8080)
    .on('listening', function() {
      console.log('Server Running: View at
        http://localhost:8080');
    });
});
```

Once that has been added, this is what our `gulpfile.js` should look like now:



```

16     .pipe(gulp.dest('dist'));
17 });
18
19 // Scripts Task
20 gulp.task('scripts', function() {
21   return gulp.src('app/js/*.js')
22     .pipe(jshint())
23     .pipe(jshint.reporter('default'))
24     .pipe(concat('all.js'))
25     .pipe(uglify())
26     .pipe(gulp.dest('dist'));
27 });
28
29 // Images Task
30 gulp.task('images', function() {
31   return gulp.src('app/img/*')
32     .pipe(imagemin())
33     .pipe(gulp.dest('dist/img'));
34 });
35
36 // Server Task
37 gulp.task('server', function() {
38   return connect().use(serve(__dirname))
39     .listen(8080)
40     .on('listening', function() {
41       console.log('Server Running: View at http://localhost:8080');
42     });
43 });

```

Line 36, Column 1 Spaces: 4 JavaScript

The first thing you will notice is that aside from our main `.task()` wrapper method we don't actually use `gulp` at all in this task. It's literally a wrapper to label and run the `node.js` code that resides within.

Let's take a moment to discuss this code to better understand what it does:

First we include our `connect()` function. Next, we will use its `.use()` method and pass it to our `serve()` module. In the `serve()` module we will pass it to the directory we wish to serve from, in our case `__dirname`, which is used by `node.js` to output the name of the directory that the currently executing script resides in. Next we assign port 8080 to listen for requests. Finally, we use the `.on()` method to check whether our server is successfully listening, and then we log a small command to announce that the server is running as expected.

Compared to the gulp tasks we've created so far, not a lot has changed. The only difference is that we are using the methods for the plain node.js module instead of gulp's built-in methods such as `.src()` and `.dest()`. That's because in this case we aren't actually using gulp to modify any data. We are only using it to label, organize, and control the use of a plain node.js module within a gulp task. The server doesn't have any use for modifying our data or using streams, it simply exists as a way to serve the files to a browser.

Finally, if you would like, you can include this task inside your default task so that this task is run by default.

When added, your default task should now look like this:

```
// Default Task
gulp.task('default', gulp.parallel('styles', 'scripts', 'images',
'server', 'watch'));
```

BrowserSync

As web developers, we spend a lot of time interacting with our browsers. Whether we are debugging our code, resizing our windows, or simply refreshing our pages, we often perform a lot of repetitive tasks in order to do our jobs.

In this section, we will explore ways to eliminate browser refreshes and make some other handy improvements to our browser experience. To do this, we will use an incredible node.js module called BrowserSync.

BrowserSync is one of the most impressive tools I have ever used. Upon first use, it will truly wow you with what it is capable of doing. Unlike similar tools that only handle browser refreshing, BrowserSync will additionally sync up every action that is performed on your pages across any device on your local network.

This process allows you to have multiple devices viewing the same project simultaneously and maintains actions, such as scrolling, in sync across them all. It's really quite impressive and can save you a ton of time when developing, especially if you're working on responsive designs.

Installing BrowserSync

To use BrowserSync, we first need to install it. The process is the same as all of the other plugins and modules that we have installed previously.

To install BrowserSync run the following command:

```
npm install browser-sync --save-dev
```

As always, we will include our `--save-dev` flag to ensure that it is added to our development dependencies list.

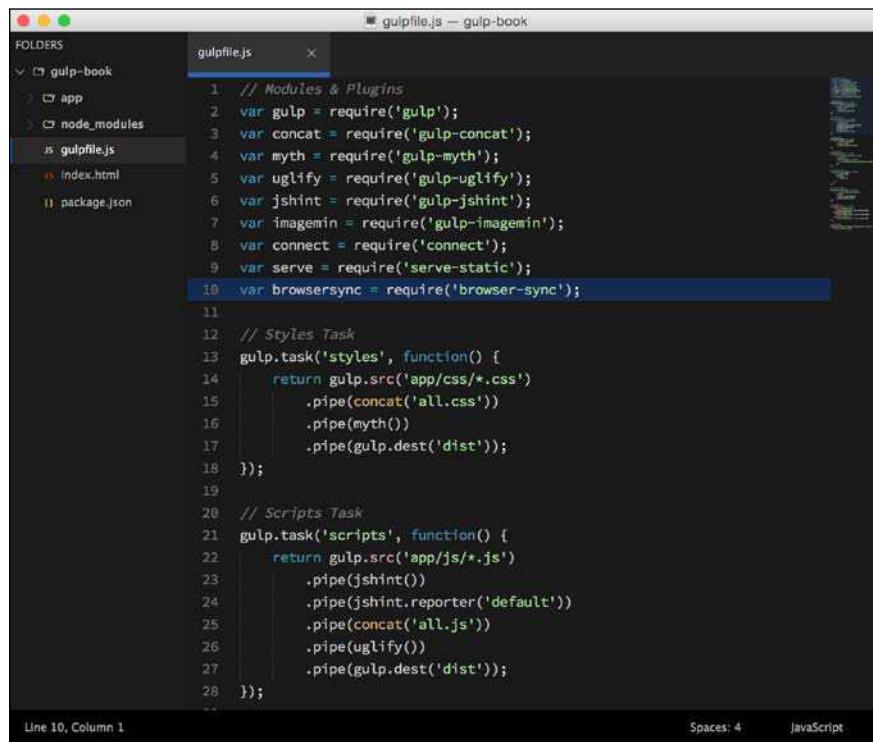
Including BrowserSync

Once installed, we can add the module to our project by adding it to our list of requires at the top of our gulpfile.

The module/plugin to be included in the code is as follows:

```
// Modules & Plugins
var gulp = require('gulp');
var concat = require('gulp-concat');
var myth = require('gulp-myth');
var uglify = require('gulp-uglify');
var jshint = require('gulp-jshint');
var imagemin = require('gulp-imagemin');
var connect = require('connect');
var serve = require('serve-static');
var browserSync = require('browser-sync'); // Added
```

After requiring BrowserSync, your gulpfile should look like the following:



```
1 // Modules & Plugins
2 var gulp = require('gulp');
3 var concat = require('gulp-concat');
4 var myth = require('gulp-myth');
5 var uglify = require('gulp-uglify');
6 var jshint = require('gulp-jshint');
7 var imagemin = require('gulp-imagemin');
8 var connect = require('connect');
9 var serve = require('serve-static');
10 var browserSync = require('browser-sync'); // Added

11
12 // Styles Task
13 gulp.task('styles', function() {
14   return gulp.src('app/css/*.css')
15     .pipe(concat('all.css'))
16     .pipe(myth())
17     .pipe(gulp.dest('dist'));
18 });
19
20 // Scripts Task
21 gulp.task('scripts', function() {
22   return gulp.src('app/js/*.js')
23     .pipe(jshint())
24     .pipe(jshint.reporter('default'))
25     .pipe(concat('all.js'))
26     .pipe(uglify())
27     .pipe(gulp.dest('dist'));
28 });
29
```

Line 10, Column 1 Spaces: 4 JavaScript

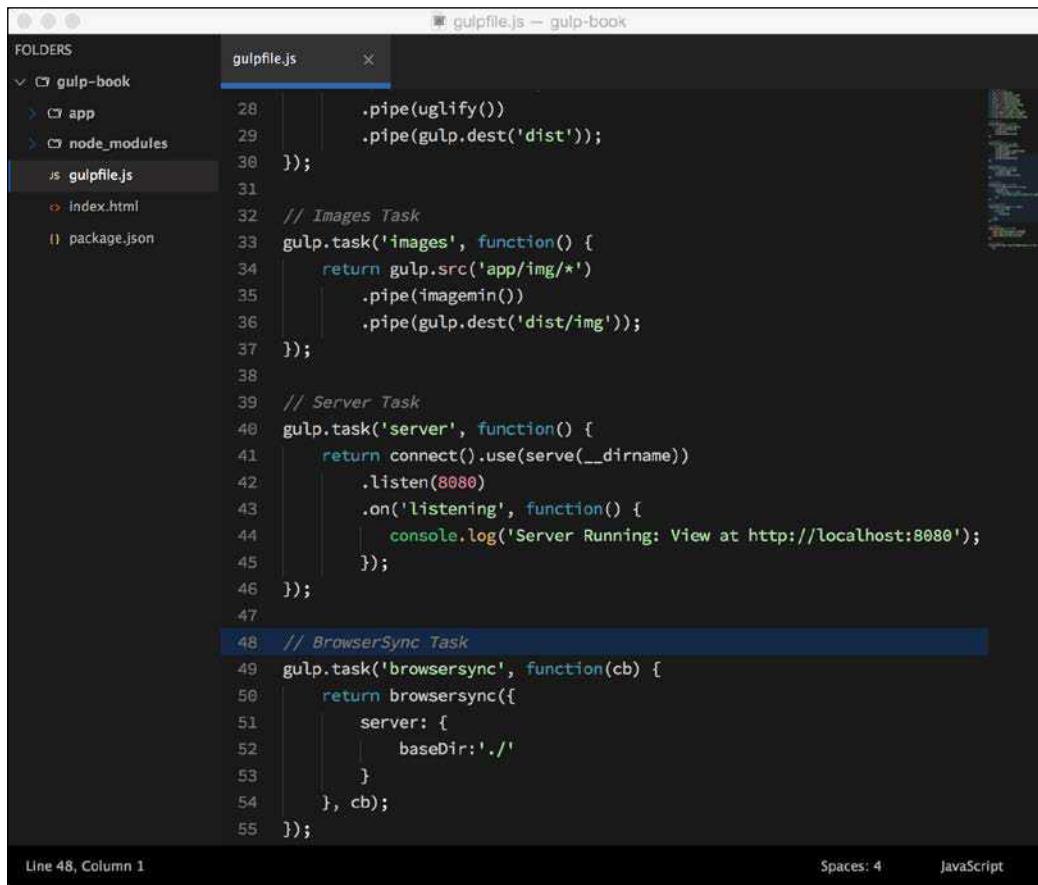
Writing the BrowserSync task

Now, let's create a small task that we can call anytime we need to communicate any changes we make to our browsers.

The code for the `BrowserSync` task is as follows:

```
gulp.task('browsersync', function(cb) {
  return browsersync({
    server: {
      baseDir:'./'
    }
  }, cb);
});
```

The following screenshot shows the completed `BrowserSync` task in our `gulpfile`:



A screenshot of a code editor showing the `gulpfile.js` file. The file contains several tasks: `images`, `server`, and `browsersync`. The `browsersync` task is highlighted with a blue background. The code editor interface includes a sidebar with project folders like `gulp-book`, `app`, and `node_modules`, and a status bar at the bottom indicating "Line 48, Column 1", "Spaces: 4", and "JavaScript".

```
gulpfile.js — gulp-book
FOLDERS
 gulp-book
  app
  node_modules
  gulpfile.js
  index.html
  package.json

28   .pipe(uglify())
29   .pipe(gulp.dest('dist'));
30 });
31
32 // Images Task
33 gulp.task('images', function() {
34   return gulp.src('app/img/*')
35     .pipe(imagemin())
36     .pipe(gulp.dest('dist/img'));
37 });
38
39 // Server Task
40 gulp.task('server', function() {
41   return connect().use(serve(__dirname))
42     .listen(8080)
43     .on('listening', function() {
44       console.log('Server Running: View at http://localhost:8080');
45     });
46 });
47
48 // BrowserSync Task
49 gulp.task('browsersync', function(cb) {
50   return browsersync({
51     server: {
52       baseDir:'./'
53     }
54   }, cb);
55 });

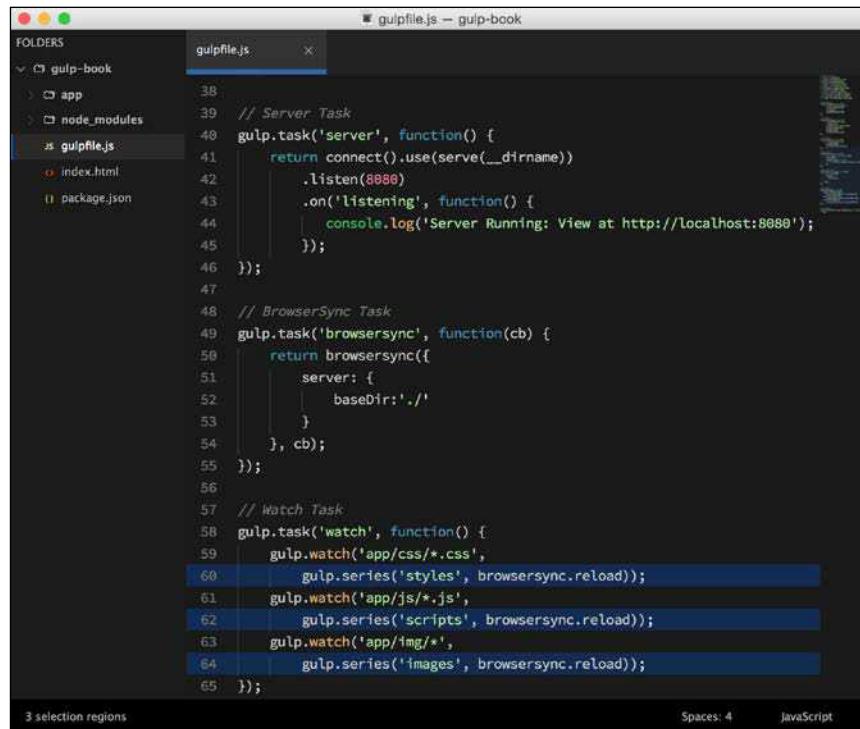
Line 48, Column 1
Spaces: 4
JavaScript
```

In this task, we have simply called our `browsersync` module and provided it with our base project directory as the location to create the server instance.

As a final step, we need to add some additional information to our `watch` task to let `BrowserSync` know when to reload our browsers:

```
// Watch Task
gulp.task('watch', function() {
    gulp.watch('app/css/*.css', gulp.series('styles',
        browsersync.reload));
    gulp.watch('app/js/*.js', gulp.series('scripts',
        browsersync.reload));
    gulp.watch('app/img/*', gulp.series('images',
        browsersync.reload));
});
```

Now, we need our `watch` methods to run two items instead of one. So, we have added in the `.series` method to execute our tasks in a specified order. In the `series` method, we will pass in our task name first, and then include a reference to the `.reload` method of our `browsersync` task. This will allow our tasks to complete before communicating any changes to our source files and instruct `BrowserSync` to refresh our browsers.



```
// Server Task
gulp.task('server', function() {
    return connect().use(serve(__dirname))
        .listen(8080)
        .on('listening', function() {
            console.log('Server Running: View at http://localhost:8080');
        });
});

// BrowserSync Task
gulp.task('browsersync', function(cb) {
    return browsersync({
        server: {
            baseDir:'./'
        },
        cb
    });
});

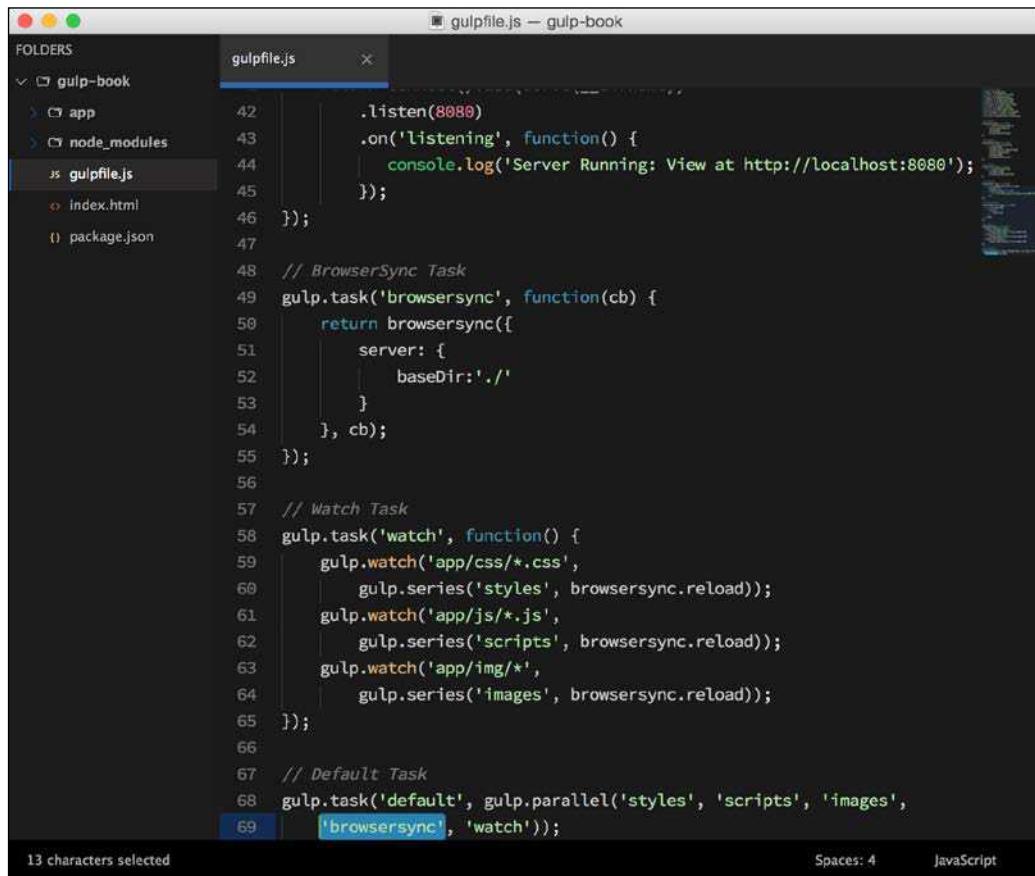
// Watch Task
gulp.task('watch', function() {
    gulp.watch('app/css/*.css',
        gulp.series('styles', browsersync.reload));
    gulp.watch('app/js/*.js',
        gulp.series('scripts', browsersync.reload));
    gulp.watch('app/img/*',
        gulp.series('images', browsersync.reload));
});
```

Using Node.js Modules for Advanced Tasks

If you would like this task to run by default, be sure that you also include it to your default task like so:

```
// Default Task
gulp.task('default', gulp.parallel('styles', 'scripts', 'images',
  'browsersync', 'watch'));
```

Once this has been added, your gulpfile should look like the following:



```
gulpfile.js — gulp-book
FOLDERS: gulp-book
app
node_modules
gulpfile.js
index.html
package.json

13 characters selected
Spaces: 4
JavaScript

// Default Task
gulp.task('default', gulp.parallel('styles', 'scripts', 'images',
  'browsersync', 'watch'));

// Server Task
gulp.task('server', function() {
  var http = require('http');
  var fs = require('fs');

  var options = {
    port: 8080,
    host: '0.0.0.1'
  };

  var server = http.createServer(function(req, res) {
    var filePath = './' + req.url;
    if (req.url === '/') {
      filePath = './index.html';
    }
    fs.readFile(filePath, function(err, data) {
      if (err) {
        res.writeHead(404);
        return res.end();
      }
      res.writeHead(200, {'Content-Type': 'text/html'});
      res.end(data);
    });
  });

  server.listen(options.port, options.host, function() {
    console.log('Server Running: View at http://localhost:8080');
  });
});

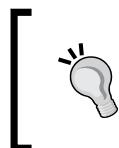
// BrowserSync Task
gulp.task('browsersync', function(cb) {
  return browsersync({
    server: {
      baseDir: './'
    }
  }, cb);
});

// Watch Task
gulp.task('watch', function() {
  gulp.watch('app/css/*.css', gulp.series('styles', browsersync.reload));
  gulp.watch('app/js/*.js', gulp.series('scripts', browsersync.reload));
  gulp.watch('app/img/*', gulp.series('images', browsersync.reload));
});
});

// Default Task
gulp.task('default', gulp.parallel('styles', 'scripts', 'images',
  'browsersync', 'watch'));
```

Going into a lot of detail about BrowserSync is really out of the scope of this book, however it's worth knowing what to expect when you run this task. As soon as gulp runs our browsersync task, it will immediately create a server and open a new browser window pointing to `http://localhost:3000`, which is the default port that BrowserSync uses. Once this has been completed, everything that runs on that page will be automatically refreshed if you update your code.

Additionally, you will be given an external URL that you can visit on other devices, such as a phone or tablet, as long as they are all on the same network. Once you have visited that URL, all of your actions will be kept in sync and any time you make changes to your code, all of the devices will refresh to show those changes automatically. It even tracks your scrolling movement on every single device, so if you decide to scroll up on your phone, the website will also scroll up on every other device, including your laptop or computer. It's an incredibly neat and helpful tool.



It is worth noting that using both a static server and BrowserSync are unnecessary as they serve a similar purpose. It's really dependent on which suits your project best. In most cases, I would suggest using BrowserSync due to the added features that it provides.



Browserify

As you have now experienced when creating a gulpfile and writing tasks, the way node.js breaks code into modules is very clean and natural. With node.js we can assign an entire module to a variable by using node.js' `require()` function.

This pattern is actually based on a specification called **CommonJS** and it is a truly fantastic way to organize and modularize code. Browserify is a tool that was created to leverage that exact same specification so that you can write all of your JavaScript code that way. Not only will you be able to modularize your own project code, but you now have the ability to use modules from npm in your non-node.js JavaScript. It's quite remarkable.

The goal of this task is to use Browserify so that we can write our JavaScript files using the CommonJS spec that node.js uses to include and modularize various pieces of our application. Additionally, we will also be able to use many other node.js modules in our projects on the client side without having to run them on a server.

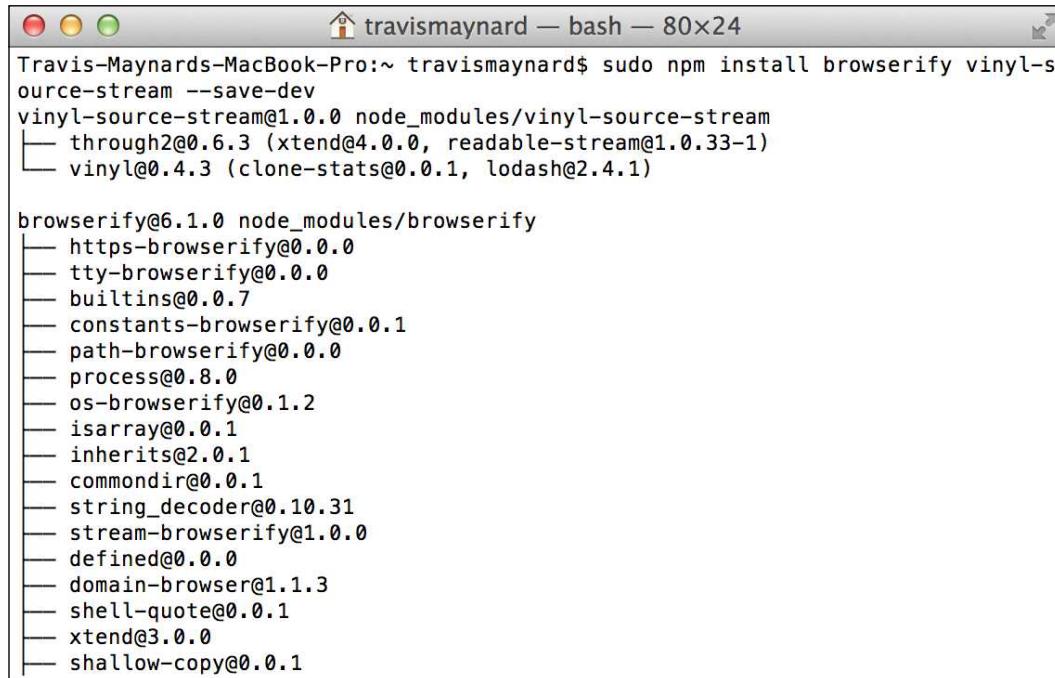
Installing modules

We will use both the `browserify` and `vinyl-source-stream` modules for this task. Many node.js modules operate using node.js streams, but gulp uses a virtual file format called `vinyl` to process files. So, to interact with modules such as Browserify, we must convert the stream into a format that we can use by including the `vinyl-source-stream` module.

To install these modules, run the following command:

```
npm install browserify vinyl-source-stream --save-dev
```

Upon running the command you should see a response like the following:



```
travismaynard — bash — 80x24
Travis-Maynards-MacBook-Pro:~ travismaynard$ sudo npm install browserify vinyl-source-stream --save-dev
vinyl-source-stream@1.0.0 node_modules/vinyl-source-stream
└── through2@0.6.3 (xtend@4.0.0, readable-stream@1.0.33-1)
    └── vinyl@0.4.3 (clone-stats@0.0.1, lodash@2.4.1)

browserify@6.1.0 node_modules/browserify
└── https-browserify@0.0.0
    ├── tty-browserify@0.0.0
    ├── builtins@0.0.7
    ├── constants-browserify@0.0.1
    ├── path-browserify@0.0.0
    ├── process@0.8.0
    ├── os-browserify@0.1.2
    ├── isarray@0.0.1
    ├── inherits@2.0.1
    ├── commondir@0.0.1
    ├── string_decoder@0.10.31
    ├── stream-browserify@1.0.0
    ├── defined@0.0.0
    ├── domain-browser@1.1.3
    ├── shell-quote@0.0.1
    ├── xtend@3.0.0
    └── shallow-copy@0.0.1
```

Including modules

Once we have installed our modules, we can add them to our gulpfile by appending them to our list of requires, like this:

```
// Modules & Plugins
var gulp = require('gulp');
var concat = require('gulp-concat');
var myth = require('gulp-myth');
var uglify = require('gulp-uglify');
var jshint = require('gulp-jshint');
var imagemin = require('gulp-imagemin');
var connect = require('connect');
var serve = require('serve-static');
var browsersync = require('browser-sync');
var browserify = require('browserify'); // Added
var source = require('vinyl-source-stream'); // Added
```

Our gulpfile should now look like the following:

```

1 // Modules & Plugins
2 var gulp = require('gulp');
3 var concat = require('gulp-concat');
4 var myth = require('gulp-myth');
5 var uglify = require('gulp-uglify');
6 var jshint = require('gulp-jshint');
7 var imagemin = require('gulp-imagemin');
8 var connect = require('connect');
9 var serve = require('serve-static');
10 var browserSync = require('browser-sync');
11 var browserify = require('browserify');
12 var source = require('vinyl-source-stream');

13
14 // Styles Task
15 gulp.task('styles', function() {
16   return gulp.src('app/css/*.css')
17     .pipe(concat('all.css'))
18     .pipe(myth())
19     .pipe(gulp.dest('dist'));
20 });
21
22 // Scripts Task
23 gulp.task('scripts', function() {
24   return gulp.src('app/js/*.js')
25     .pipe(jshint())
26     .pipe(jshint.reporter('default'))
27     .pipe(concat('all.js'))
28     .pipe(uglify())
29 });

Spaces: 4 JavaScript

```

Writing the Browserify task

As with all of our other tasks, we always start with our main task wrapper method and provide our task with a name. In this task, we will blend new methods from our Browserify module with some of gulp's methods that you are already familiar with.

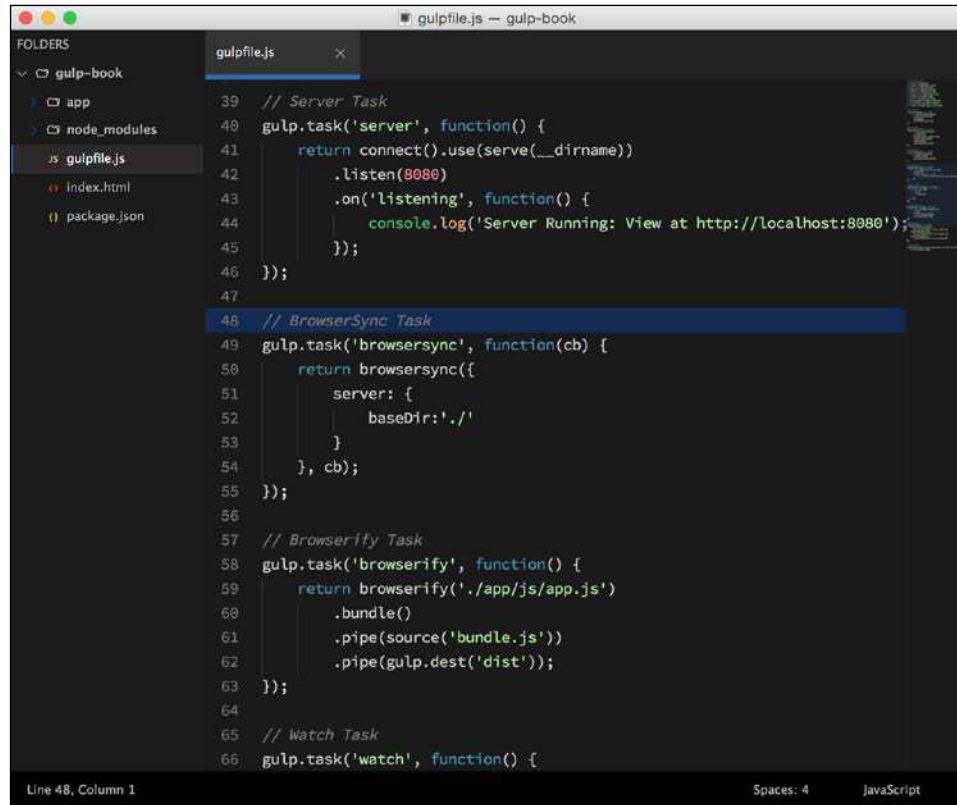
The code to run Browserify as a gulp task look like this:

```

gulp.task('browserify', function() {
  return browserify('./app/js/app.js')
    .bundle()
    .pipe(source('bundle.js'))
    .pipe(gulp.dest('dist'));
});

```

With the Browserify task added, our gulpfile will look like this:



```
gulpfile.js — gulp-book
FOLDERS
  gulp-book
    app
    node_modules
      js
        gulpfile.js
      index.html
    package.json
Line 48, Column 1
Spaces: 4
JavaScript
1 // Server Task
2 gulp.task('server', function() {
3   return connect().use(serve(__dirname))
4     .listen(8080)
5     .on('listening', function() {
6       console.log('Server Running: View at http://localhost:8080');
7     });
8 });
9
10 // BrowserSync Task
11 gulp.task('browsersync', function(cb) {
12   return browsersync({
13     server: {
14       baseDir:'./'
15     },
16     cb,
17   });
18
19 // Browserify Task
20 gulp.task('browserify', function() {
21   return browserify('./app/js/app.js')
22     .bundle()
23     .pipe(source('bundle.js'))
24     .pipe(gulp.dest('dist'));
25 });
26
27 // Watch Task
28 gulp.task('watch', function() {
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
259
260
261
262
263
264
265
266
267
268
269
269
270
271
272
273
274
275
276
277
278
279
279
280
281
282
283
284
285
286
287
287
288
289
289
290
291
292
293
294
295
296
297
297
298
299
299
300
301
302
303
304
305
306
307
307
308
309
309
310
311
311
312
312
313
313
314
314
315
315
316
316
317
317
318
318
319
319
320
320
321
321
322
322
323
323
324
324
325
325
326
326
327
327
328
328
329
329
330
330
331
331
332
332
333
333
334
334
335
335
336
336
337
337
338
338
339
339
340
340
341
341
342
342
343
343
344
344
345
345
346
346
347
347
348
348
349
349
350
350
351
351
352
352
353
353
354
354
355
355
356
356
357
357
358
358
359
359
360
360
361
361
362
362
363
363
364
364
365
365
366
366
367
367
368
368
369
369
370
370
371
371
372
372
373
373
374
374
375
375
376
376
377
377
378
378
379
379
380
380
381
381
382
382
383
383
384
384
385
385
386
386
387
387
388
388
389
389
390
390
391
391
392
392
393
393
394
394
395
395
396
396
397
397
398
398
399
399
400
400
401
401
402
402
403
403
404
404
405
405
406
406
407
407
408
408
409
409
410
410
411
411
412
412
413
413
414
414
415
415
416
416
417
417
418
418
419
419
420
420
421
421
422
422
423
423
424
424
425
425
426
426
427
427
428
428
429
429
430
430
431
431
432
432
433
433
434
434
435
435
436
436
437
437
438
438
439
439
440
440
441
441
442
442
443
443
444
444
445
445
446
446
447
447
448
448
449
449
450
450
451
451
452
452
453
453
454
454
455
455
456
456
457
457
458
458
459
459
460
460
461
461
462
462
463
463
464
464
465
465
466
466
467
467
468
468
469
469
470
470
471
471
472
472
473
473
474
474
475
475
476
476
477
477
478
478
479
479
480
480
481
481
482
482
483
483
484
484
485
485
486
486
487
487
488
488
489
489
490
490
491
491
492
492
493
493
494
494
495
495
496
496
497
497
498
498
499
499
500
500
501
501
502
502
503
503
504
504
505
505
506
506
507
507
508
508
509
509
510
510
511
511
512
512
513
513
514
514
515
515
516
516
517
517
518
518
519
519
520
520
521
521
522
522
523
523
524
524
525
525
526
526
527
527
528
528
529
529
530
530
531
531
532
532
533
533
534
534
535
535
536
536
537
537
538
538
539
539
540
540
541
541
542
542
543
543
544
544
545
545
546
546
547
547
548
548
549
549
550
550
551
551
552
552
553
553
554
554
555
555
556
556
557
557
558
558
559
559
560
560
561
561
562
562
563
563
564
564
565
565
566
566
567
567
568
568
569
569
570
570
571
571
572
572
573
573
574
574
575
575
576
576
577
577
578
578
579
579
580
580
581
581
582
582
583
583
584
584
585
585
586
586
587
587
588
588
589
589
590
590
591
591
592
592
593
593
594
594
595
595
596
596
597
597
598
598
599
599
600
600
601
601
602
602
603
603
604
604
605
605
606
606
607
607
608
608
609
609
610
610
611
611
612
612
613
613
614
614
615
615
616
616
617
617
618
618
619
619
620
620
621
621
622
622
623
623
624
624
625
625
626
626
627
627
628
628
629
629
630
630
631
631
632
632
633
633
634
634
635
635
636
636
637
637
638
638
639
639
640
640
641
641
642
642
643
643
644
644
645
645
646
646
647
647
648
648
649
649
650
650
651
651
652
652
653
653
654
654
655
655
656
656
657
657
658
658
659
659
660
660
661
661
662
662
663
663
664
664
665
665
666
666
667
667
668
668
669
669
670
670
671
671
672
672
673
673
674
674
675
675
676
676
677
677
678
678
679
679
680
680
681
681
682
682
683
683
684
684
685
685
686
686
687
687
688
688
689
689
690
690
691
691
692
692
693
693
694
694
695
695
696
696
697
697
698
698
699
699
700
700
701
701
702
702
703
703
704
704
705
705
706
706
707
707
708
708
709
709
710
710
711
711
712
712
713
713
714
714
715
715
716
716
717
717
718
718
719
719
720
720
721
721
722
722
723
723
724
724
725
725
726
726
727
727
728
728
729
729
730
730
731
731
732
732
733
733
734
734
735
735
736
736
737
737
738
738
739
739
740
740
741
741
742
742
743
743
744
744
745
745
746
746
747
747
748
748
749
749
750
750
751
751
752
752
753
753
754
754
755
755
756
756
757
757
758
758
759
759
760
760
761
761
762
762
763
763
764
764
765
765
766
766
767
767
768
768
769
769
770
770
771
771
772
772
773
773
774
774
775
775
776
776
777
777
778
778
779
779
780
780
781
781
782
782
783
783
784
784
785
785
786
786
787
787
788
788
789
789
790
790
791
791
792
792
793
793
794
794
795
795
796
796
797
797
798
798
799
799
800
800
801
801
802
802
803
803
804
804
805
805
806
806
807
807
808
808
809
809
810
810
811
811
812
812
813
813
814
814
815
815
816
816
817
817
818
818
819
819
820
820
821
821
822
822
823
823
824
824
825
825
826
826
827
827
828
828
829
829
830
830
831
831
832
832
833
833
834
834
835
835
836
836
837
837
838
838
839
839
840
840
841
841
842
842
843
843
844
844
845
845
846
846
847
847
848
848
849
849
850
850
851
851
852
852
853
853
854
854
855
855
856
856
857
857
858
858
859
859
860
860
861
861
862
862
863
863
864
864
865
865
866
866
867
867
868
868
869
869
870
870
871
871
872
872
873
873
874
874
875
875
876
876
877
877
878
878
879
879
880
880
881
881
882
882
883
883
884
884
885
885
886
886
887
887
888
888
889
889
890
890
891
891
892
892
893
893
894
894
895
895
896
896
897
897
898
898
899
899
900
900
901
901
902
902
903
903
904
904
905
905
906
906
907
907
908
908
909
909
910
910
911
911
912
912
913
913
914
914
915
915
916
916
917
917
918
918
919
919
920
920
921
921
922
922
923
923
924
924
925
925
926
926
927
927
928
928
929
929
930
930
931
931
932
932
933
933
934
934
935
935
936
936
937
937
938
938
939
939
940
940
941
941
942
942
943
943
944
944
945
945
946
946
947
947
948
948
949
949
950
950
951
951
952
952
953
953
954
954
955
955
956
956
957
957
958
958
959
959
960
960
961
961
962
962
963
963
964
964
965
965
966
966
967
967
968
968
969
969
970
970
971
971
972
972
973
973
974
974
975
975
976
976
977
977
978
978
979
979
980
980
981
981
982
982
983
983
984
984
985
985
986
986
987
987
988
988
989
989
990
990
991
991
992
992
993
993
994
994
995
995
996
996
997
997
998
998
999
999
1000
1000
1001
1001
1002
1002
1003
1003
1004
1004
1005
1005
1006
1006
1007
1007
1008
1008
1009
1009
1010
1010
1011
1011
1012
1012
1013
1013
1014
1014
1015
1015
1016
1016
1017
1017
1018
1018
1019
1019
1020
1020
1021
1021
1022
1022
1023
1023
1024
1024
1025
1025
1026
1026
1027
1027
1028
1028
1029
1029
1030
1030
1031
1031
1032
1032
1033
1033
1034
1034
1035
1035
1036
1036
1037
1037
1038
1038
1039
1039
1040
1040
1041
1041
1042
1042
1043
1043
1044
1044
1045
1045
1046
1046
1047
1047
1048
1048
1049
1049
1050
1050
1051
1051
1052
1052
1053
1053
1054
1054
1055
1055
1056
1056
1057
1057
1058
1058
1059
1059
1060
1060
1061
1061
1062
1062
1063
1063
1064
1064
1065
1065
1066
1066
1067
1067
1068
1068
1069
1069
1070
1070
1071
1071
1072
1072
1073
1073
1074
1074
1075
1075
1076
1076
1077
1077
1078
1078
1079
1079
1080
1080
1081
1081
1082
1082
1083
1083
1084
1084
1085
1085
1086
1086
1087
1087
1088
1088
1089
1089
1090
1090
1091
1091
1092
1092
1093
1093
1094
1094
1095
1095
1096
1096
1097
1097
1098
1098
1099
1099
1100
1100
1101
1101
1102
1102
1103
1103
1104
1104
1105
1105
1106
1106
1107
1107
1108
1108
1109
1109
1110
1110
1111
1111
1112
1112
1113
1113
1114
1114
1115
1115
1116
1116
1117
1117
1118
1118
1119
1119
1120
1120
1121
1121
1122
1122
1123
1123
1124
1124
1125
1125
1126
1126
1127
1127
1128
1128
1129
1129
1130
1130
1131
1131
1132
1132
1133
1133
1134
1134
1135
1135
1136
1136
1137
1137
1138
1138
1139
1139
1140
1140
1141
1141
1142
1142
1143
1143
1144
1144
1145
1145
1146
1146
1147
1147
1148
1148
1149
1149
1150
1150
1151
1151
1152
1152
1153
1153
1154
1154
1155
1155
1156
1156
1157
1157
1158
1158
1159
1159
1160
1160
1161
1161
1162
1162
1163
1163
1164
1164
1165
1165
1166
1166
1167
1167
1168
1168
1169
1169
1170
1170
1171
1171
1172
1172
1173
1173
1174
1174
1175
1175
1176
1176
1177
1177
1178
1178
1179
1179
1180
1180
1181
1181
1182
1182
1183
1183
1184
1184
1185
1185
1186
1186
1187
1187
1188
1188
1189
1189
1190
1190
1191
1191
1192
1192
1193
1193
1194
1194
1195
1195
1196
1196
1197
1197
1198
1198
1199
1199
1200
1200
1201
1201
1202
1202
1203
1203
1204
1204
1205
1205
1206
1206
1207
1207
1208
1208
1209
1209
1210
1210
1211
1211
1212
1212
1213
1213
1214
1214
1215
1215
1216
1216
1217
1217
1218
1218
1219
1219
1220
1220
1221
1221
1222
1222
1223
1223
1224
1224
1225
1225
1226
1226
1227
1227
1228
1228
1229
1229
1230
1230
1231
1231
1232
1232
1233
1233
1234
1234
1235
1235
1236
1236
1237
1237
1238
1238
1239
1239
1240
1240
1241
1241
1242
1242
1243
1243
1244
1244
1245
1245
1246
1246
1247
1247
1248
1248
1249
1249
1250
1250
1251
1251
1252
1252
1253
1253
1254
1254
1255
1255
1256
1256
1257
1257
1258
1258
1259
1259
1260
1260
1261
1261
1262
1262
1263
1263
1264
1264
1265
1265
1266
1266
1267
1267
1268
1268
1269
1269
1270
1270
1271
1271
1272
1272
1273
1273
1274
1274
1275
1275
1276
1276
1277
1277
1278
1278
1279
1279
1280
1280
1281
1281
1282
1282
1283
1283
1284
1284
1285
1285
1286
1286
1287
1287
1288
1288
1289
1289
1290
1290
1291
1291
1292
1292
1293
1293
1294
1294
1295
1295
1296
1296
1297
1297
1298
1298
1299
1299
1300
1300
1301
1301
1302
1302
1303
1303
1304
1304
1305
1305
1306
1306
1307
1307
1308
1308
1309
1309
1310
1310
1311
1311
1312
1312
1313
1313
1314
1314
1315
1315
1316
1316
1317
1317
1318
1318
1319
1319
1320
1320
1321
1321
1322
1322
1323
1323
1324
1324
1325
1325
1326
1326
1327
1327
1328
1328
1329
1329
1330
1330
1331
1331
1332
1332
1333
1333
1334
1334
1335
1335
1336
1336
1337
1337
1338
1338
1339
1339
1340
1340
1341
1341
1342
1342
1343
1343
1344
1344
1345
1345
1346
1346
1347
1347
1348
1348
1349
1349
1350
1350
1351
1351
1352
1352
1353
1353
1354
1354
1355
1355
1356
1356
1357
1357
1358
1358
1359
1359
1360
1360
1361
1361
1362
1362
1363
1363
1364
1364
1365
1365
1366
1366
1367
1367
1368
1368
1369
1369
1370
1370
1371
1371
1372
1372
1373
1373
1374
1374
1375
1375
1376
1376
1377
1377
1378
1378
1379
1379
1380
1380
1381
1381
1382
1382
1383
1383
1384
1384
1385
1385
1386
1386
1387
1387
1388
1388
1389
1389
1390
1390
1391
1391
1392
1392
1393
1393
1394
1394
1395
1395
1396
1396
1397
1397
1398
1398
1399
1399
1400
1400
1401
1401
1402
1402
1403
1403
1404
1404
1405
1405
1406
1406
1407
1407
1408
1408
1409
1409
1410
1410
1411
1411
1412
1412
1413
1413
1414
1414
1415
1415
1416
1416
1417
1417
1418
1418
1419
1419
1420
1420
1421
1421
1422
1422
1423
1423
1424
1424
1425
1425
1426
1426
1427
1427
1428
1428
1429
1429
1430
1430
1431
1431
1432
1432
1433
```

The only really confusing portion of this task is understanding the difference between node.js streams and gulp streams. Beyond that, the task runs somewhat like our original gulp tasks, which we created in the previous chapter. Knowing when and how to work around the differences in streams will make your life a lot easier when using plain node.js modules inside of gulp tasks.



As with all tasks, if you would like this to be run by default or have it run upon file changes, be sure that you also include it in your default and/or watch tasks as we have done throughout the book.

While this task sets you up to use Browserify inside gulp, you should take some time to really understand how Browserify works and how to make use of it in your projects. To learn more about Browserify, be sure to check out the official site at <http://browserify.org>.

Summary

In this chapter, we took a look at three advanced tasks using plain node.js modules instead of gulp plugins. Our first task creates a simple static server so that we can view our project in a browser. The second assists us by automatically refreshing our browser and keeping our actions in sync across multiple devices as we work. Finally, the third task allows us to use the node.js/CommonJS spec to modularize our client-side code and write it as if it is a node.js application.

In the next chapter, we will continue to improve our gulpfile by taking a look at some valuable tips and resolve some common issues that are easy to run into while using gulp.

5

Resolving Issues

By this point, we have highlighted the many ways through which gulp will improve your workflow and help you deliver more optimized and performant code. As with all software, there are some quirks that you may run into while using gulp that could make the experience less than perfect.

In this chapter, we will explore some of the troubles that you may run into while using gulp and introduce you to solutions to get around them.

Handling errors

One of the biggest problems that I encountered when first learning gulp was how to handle it when something failed. Unfortunately, gulp doesn't have a clean way to handle errors, and when failures do occur, it doesn't handle them very gracefully. For example, let's say we have our watch task running in the background as we are editing a Sass file. We're typing away and styling our website, but we accidentally hit a key that the Sass plugin wasn't expecting. Instead of failing through the code and just displaying the page break at that moment, gulp will simply throw an error and the watch task will stop running.

The main problem with this is that you may not actually realize that gulp has stopped and you will continue working on your code, only to realize moments later that all of your changes aren't being reflected on the page you are working on. It can cause a lot of confusion and end up wasting a lot of time until you know when to expect it.

The gulp team acknowledge that this is one of the pain points when using gulp and they realize the importance of improving it. Fortunately, they are considering this issue as one of the highest priorities for future development. There are plans to include improved error handling in the upcoming versions of gulp. However, until then we can still improve our error handling by introducing a new gulp plugin called `gulp-plumber`.

Gulp-plumber was created as a stop-gap to give us more control over handling errors in our tasks. Let's take a look at how we can include it in our styles task that we created in *Chapter 3, Performing Tasks with Gulp*.

Installing gulp-plumber

Before we can begin using the plugin, we need to install it and save it to our development dependencies.

The command for installing gulp-plumber is as follows:

```
npm install gulp-plumber --save-dev
```

Upon running this command, you will see a response like the following one:

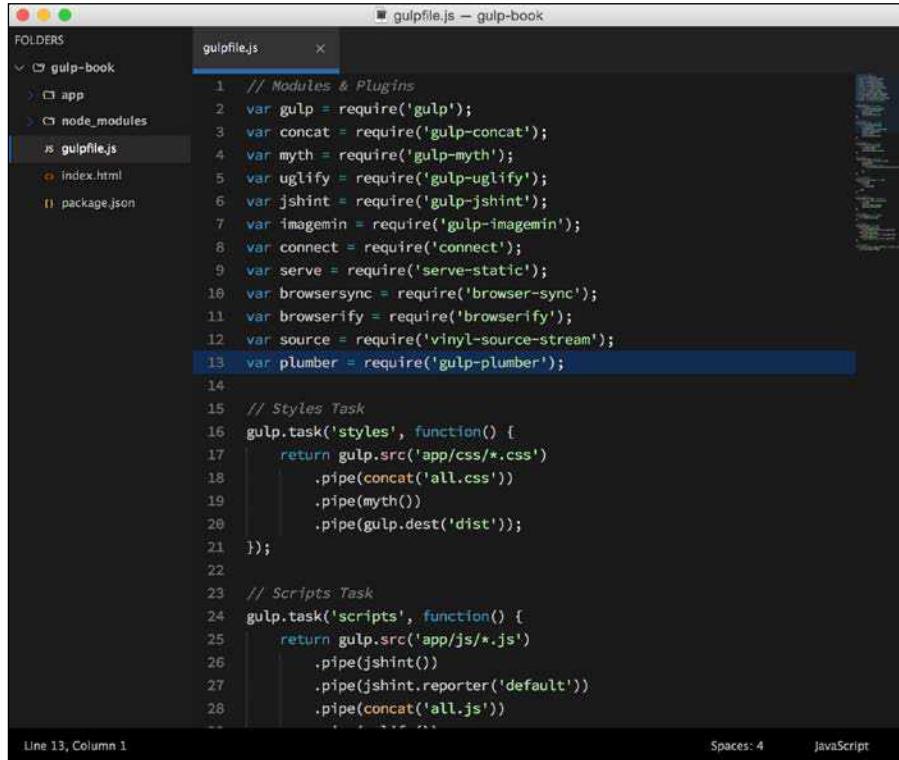
```
Travis-Maynards-MacBook-Pro:gulp-book travismaynard$ sudo npm install gulp-plumber --save-dev
npm WARN package.json gulp-book@1.0.0 No repository field.
npm WARN package.json gulp-book@1.0.0 No README data
gulp-plumber@0.6.6 node_modules/gulp-plumber
  └── through2@0.6.3 (xtend@4.0.0, readable-stream@1.0.33)
    ├── gulp-util@3.0.1 (lodash._reinterpolate@2.4.1, dateformat@1.0.8, minimist@1.1.0, lodash@2.4.1, vinyl@0.4.3, multipipe@0.1.1, chalk@0.5.1, lodash.template@2.4.1)
Travis-Maynards-MacBook-Pro:gulp-book travismaynard$
```

Including gulp-plumber

After installation, we must add it to our list of requirements to include it in our gulpfile:

```
// Modules & Plugins
var gulp = require('gulp');
var concat = require('gulp-concat');
var myth = require('gulp-myth');
var uglify = require('gulp-uglify');
var jshint = require('gulp-jshint');
var imagemin = require('gulp-imagemin');
var connect = require('connect');
var serve = require('serve-static');
var browsersync = require('browser-sync');
var browserify = require('browserify');
var source = require('vinyl-source-stream');
var plumber = require('gulp-plumber'); // Added
```

With this addition, your gulpfile should look like the following screenshot:



```
// Modules & Plugins
var gulp = require('gulp');
var concat = require('gulp-concat');
var myth = require('gulp-myth');
var uglify = require('gulp-uglify');
var jshint = require('gulp-jshint');
var imagemin = require('gulp-imagemin');
var connect = require('connect');
var serve = require('serve-static');
var browsersync = require('browser-sync');
var browserify = require('browserify');
var source = require('vinyl-source-stream');
var plumber = require('gulp-plumber');

// Styles Task
gulp.task('styles', function() {
  return gulp.src('app/css/*.css')
    .pipe(concat('all.css'))
    .pipe(myth())
    .pipe(gulp.dest('dist'));
});

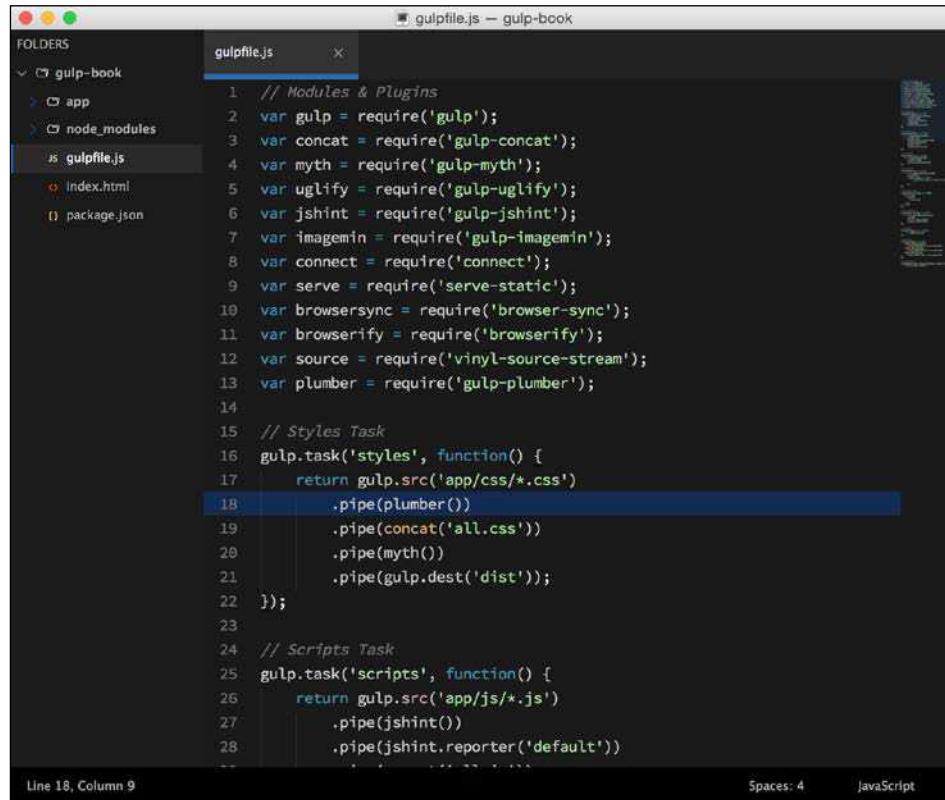
// Scripts Task
gulp.task('scripts', function() {
  return gulp.src('app/js/*.js')
    .pipe(jshint())
    .pipe(jshint.reporter('default'))
    .pipe(concat('all.js'))
});
```

Now that our plugin has been installed and included, let's take a look at how we can use it within our tasks:

```
// Styles Task
gulp.task('styles', function() {
  return gulp.src('app/css/*.css')
    .pipe(plumber())
    .pipe(concat('all.css'))
    .pipe(myth())
    .pipe(gulp.dest('dist'));
});
```

Resolving Issues

Here is an example of what this looks like in our gulpfile:



```
// Modules & Plugins
var gulp = require('gulp');
var concat = require('gulp-concat');
var myth = require('gulp-myth');
var uglify = require('gulp-uglify');
var jshint = require('gulp-jshint');
var imagemin = require('gulp-imagemin');
var connect = require('connect');
var serve = require('serve-static');
var browsersync = require('browser-sync');
var browserify = require('browserify');
var source = require('vinyl-source-stream');
var plumber = require('gulp-plumber');

// Styles Task
gulp.task('styles', function() {
  return gulp.src('app/css/*.css')
    .pipe(plumber())
    .pipe(concat('all.css'))
    .pipe(myth())
    .pipe(gulp.dest('dist'));
});

// Scripts Task
gulp.task('scripts', function() {
  return gulp.src('app/js/*.js')
    .pipe(jshint())
    .pipe(jshint.reporter('default'))
});
```

At its most basic, all that is really needed to benefit from `gulp-plumber` is to include it in the first pipe of your pipechain. In this example, it will keep `gulp.watch()` from crashing when it encounters an error and it will log error information to the console.

The only problem is that in many cases, you might not even realize an error has occurred. To remedy this, we can use an additional node module called `beeper` that will provide us with an audible alert when an error has occurred.

Installing beeper

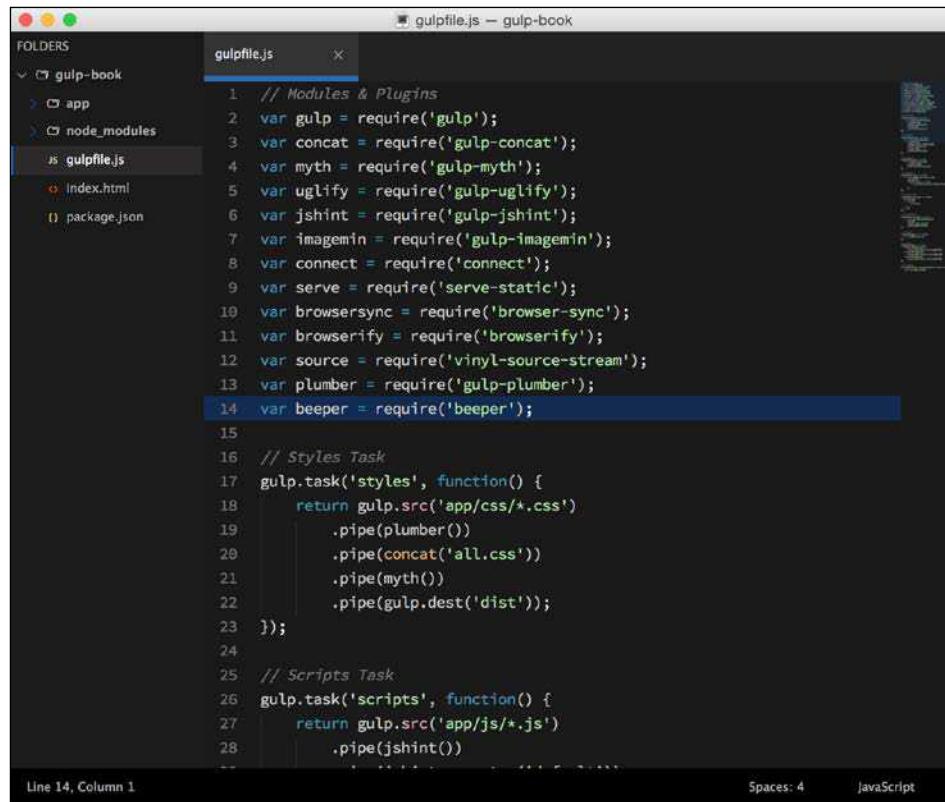
As always, we must first install the plugin via npm:

```
npm install beeper --save-dev
```

Including beeper

Once the plugin has been installed, we must add it to our list of requirements to include it into our gulpfile.

```
// Modules & Plugins
var gulp = require('gulp');
var concat = require('gulp-concat');
var myth = require('gulp-myth');
var uglify = require('gulp-uglify');
var jshint = require('gulp-jshint');
var imagemin = require('gulp-imagemin');
var connect = require('connect');
var serve = require('serve-static');
var browsersync = require('browser-sync');
var browserify = require('browserify');
var source = require('vinyl-source-stream');
var plumber = require('gulp-plumber');
var beeper = require('beeper'); // Added
```



The screenshot shows a Mac OS X desktop environment with a code editor window open. The title bar says "gulpfile.js — gulp-book". The left sidebar shows project structure with folders "gulp-book", "app", "node_modules", and "gulpfile.js" selected. The main editor area contains the Gulp script code. The line "var beeper = require('beeper');" is highlighted with a blue selection bar. The status bar at the bottom shows "Line 14, Column 1", "Spaces: 4", and "JavaScript".

```
// Modules & Plugins
var gulp = require('gulp');
var concat = require('gulp-concat');
var myth = require('gulp-myth');
var uglify = require('gulp-uglify');
var jshint = require('gulp-jshint');
var imagemin = require('gulp-imagemin');
var connect = require('connect');
var serve = require('serve-static');
var browsersync = require('browser-sync');
var browserify = require('browserify');
var source = require('vinyl-source-stream');
var plumber = require('gulp-plumber');
var beeper = require('beeper');

// Styles Task
gulp.task('styles', function() {
  return gulp.src('app/css/*.css')
    .pipe(plumber())
    .pipe(concat('all.css'))
    .pipe(myth())
    .pipe(gulp.dest('dist'));
});

// Scripts Task
gulp.task('scripts', function() {
  return gulp.src('app/js/*.js')
    .pipe(jshint())
```

Writing an error helper function

Next, we will write a simple error helper function that we can pass into `gulp-plumber` to customize how we are notified of errors.:

```
// Error Helper
function onError(err) {
    beeper();
    console.log(err);
}
```

After including this helper function, our gulpfile will look like the following screenshot:

```
// Modules & Plugins
var gulp = require('gulp');
var concat = require('gulp-concat');
var myth = require('gulp-myth');
var uglify = require('gulp-uglify');
var jshint = require('gulp-jshint');
var imagemin = require('gulp-imagemin');
var connect = require('connect');
var serve = require('serve-static');
var browsersync = require('browser-sync');
var browserify = require('browserify');
var source = require('vinyl-source-stream');
var plumber = require('gulp-plumber');
var beeper = require('beeper');

// Error Helper
function onError(err) {
    beeper();
    console.log(err);
}

// Styles Task
gulp.task('styles', function() {
    return gulp.src('app/css/*.css')
        .pipe(plumber())
        .pipe(concat('all.css'))
        .pipe(myth())
        .pipe(gulp.dest('dist'));
```

When this function is run, it will play a system sound to alert us using the `beeper` plugin and then log the error to our console. Now, let's include it in the `plumber()` pipe as an option, so that when `gulp-plumber` finds an error it will use our helper function instead of its default functionality:

```
// Styles Task
gulp.task('styles', function() {
    return gulp.src('app/css/*.css')
        .pipe(plumber({
            errorHandler: onError
        }))
        .pipe(concat('all.css'))
        .pipe(myth())
        .pipe(gulp.dest('dist'));
});
```

The next screenshot shows code for the styles task:

```
18 // Error Helper
19 function onError(err) {
20     beeper();
21     console.log(err);
22 }
23
24 // Styles Task
25 gulp.task('styles', function() {
26     return gulp.src('app/css/*.css')
27         .pipe(plumber({
28             errorHandler: onError
29         }))
30         .pipe(concat('all.css'))
31         .pipe(myth())
32         .pipe(gulp.dest('dist'));
33 });
34
35 // Scripts Task
36 gulp.task('scripts', function() {
37     return gulp.src('app/js/*.js')
38         .pipe(sourcemaps.init())
39         .pipe(jshint())
40         .pipe(jshint.reporter('default'))
41         .pipe(concat('all.js'))
42         .pipe(uglify())
43         .pipe(sourcemaps.write())
44         .pipe(gulp.dest('dist'));
45 });
```

2 selection regions Spaces: 4 JavaScript

In the preceding code, we can simply pass our new `onError()` helper function to the `errorHandler` option that is built into `gulp-plumber`. Now, we can add any additional functionality we need to our `onError()` helper function and we will receive audible feedback when something goes wrong inside our tasks.

Source ordering

Another common issue that new gulp users face is the way in which the files are ordered when they are processed. By default, each file in will be processed in order, based on its filename, unless specified otherwise. So, for example, when you are concatenating your CSS into a single file you will need to make sure that your normalized or reset styles are processed first.

To get around this, you can actually change the file names of your source files by prepending numbers to them in the order that you would like them to be processed. So, for example, if you need a `normalize.css` file to render before an `abc.css` file, you can rename those files `1-normalize.css` and `2-abc.css` respectively.

However, there are better ways to do this, and my personal favorite is to create an array at the beginning of the `gulpfile` so you can clearly order your files however you like. It's clean, simple, and easy to maintain.

Take the following code for example:

```
var cssFiles = ['assets/css/normalize.css', 'assets/css/abc.css'];
gulp.src('styles', function () {
    return gulp.src(cssFiles) // Pass in the array.
        .pipe(concat('site.css'))
        .pipe(gulp.dest('dist'));
});
```

But what if you have a large number of files and you only need to make sure that one of them is included first? Manually inserting every single one of those file paths into an array is not useful or easily maintainable, it's just time consuming and tedious.

The great news is that you can actually use globs in addition to explicit paths in your array. Gulp is smart enough to not process the same file twice. So, instead of specifying the order for every single file in the array, you can do something like this:

```
var cssFiles = ['assets/css/normalize.css', 'assets/css/*.css'];
gulp.src('styles', function () {
    return gulp.src(cssFiles) // Pass in the array.
        .pipe(concat('site.css'))
        .pipe(gulp.dest('dist'));
});
```

This will ensure that our `normalize.css` file is included first, and then it will include every other CSS file without including `normalize.css` twice in your concatenated code.

Project cleanup

Generating and processing files is great, but there may come a time when you or your teammates need to simply clear out the files that you have processed and start anew.

To do so, we are going to create another task that will clean out any processed files from our `dist` directory. To do this, we are going to use a node module called `del`, which will allow us to target multiple files and use globs in our file paths.

Installing the del module

Install the `del` module using npm and then save it to your list of development dependencies with the `--save-dev` flag:

```
npm install del --save-dev
```

The following screenshot shows the installation of the `del` module:

```
Travis-Maynards-MacBook-Pro:gulp-book travismaynard$ sudo npm install del --save-dev
npm WARN package.json gulp-book@1.0.0 No repository field.
npm WARN package.json gulp-book@1.0.0 No README data
del@0.1.3 node_modules/del
├── is-path-cwd@1.0.0
├── rimraf@2.2.8
├── each-async@1.1.0 (onetime@1.0.0, setimmediate@1.0.2)
├── is-path-in-cwd@1.0.0 (is-path-inside@1.0.0)
└── globby@0.1.1 (array-differ@0.1.0, async@0.9.0, array-union@0.1.0, glob@4.0.6)
Travis-Maynards-MacBook-Pro:gulp-book travismaynard$
```

Including the del module

Once the module has been installed you must add it to your list of required modules at the top of your `gulpfile`:

```
// Modules & Plugins
var gulp = require('gulp');
var concat = require('gulp-concat');
var myth = require('gulp-myth');
var uglify = require('gulp-uglify');
var jshint = require('gulp-jshint');
var imagemin = require('gulp-imagemin');
```

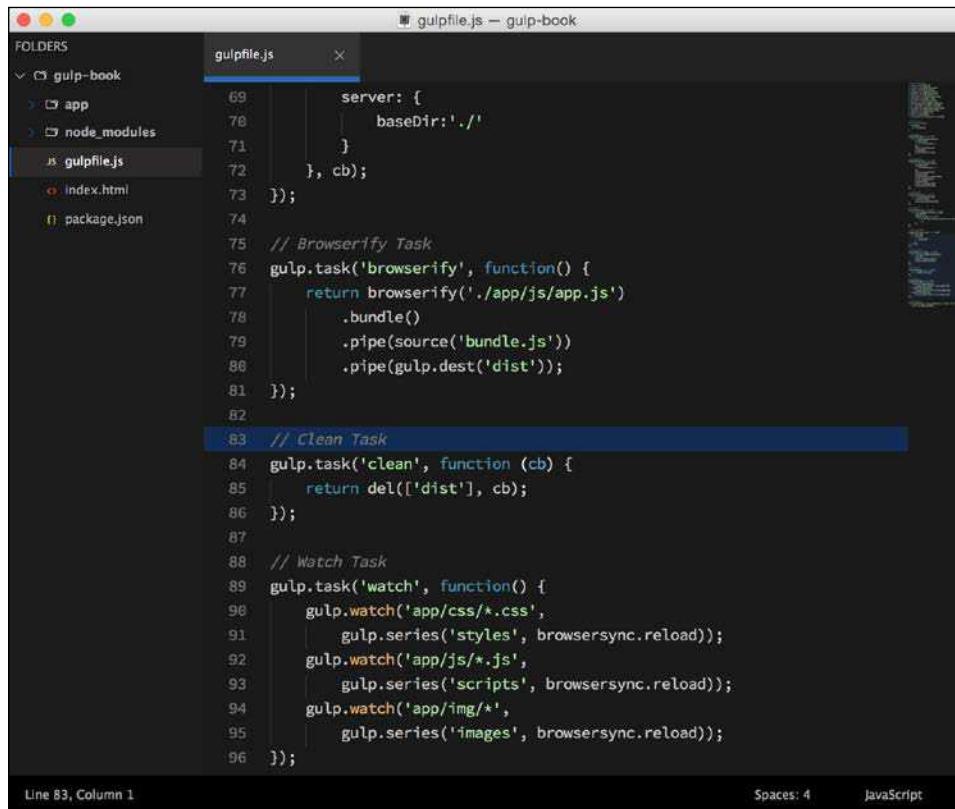
```
var connect = require('connect');
var serve = require('serve-static');
var browsersync = require('browser-sync');
var browserify = require('browserify');
var source = require('vinyl-source-stream');
var plumber = require('gulp-plumber');
var util = require('beeper');
var del = require('del'); // Added
```

Writing a clean task

One way we can use this is by deleting an entire folder altogether. So, as an example, we could delete an entire folder, such as the `dist` directory, by creating a `clean` task:

```
gulp.task('clean', function (cb) {
  del(['dist'], cb);
});
```

The following screenshot shows the newly created clean task in our `gulpfile.js`:



```
FOLDERS
gulp-book
  app
  node_modules
  gulpfile.js
  index.html
  package.json

gulpfile.js — gulp-book
server: {
  baseDir:'./'
},
cb);
});

// Browserify Task
gulp.task('browserify', function() {
  return browserify('./app/js/app.js')
    .bundle()
    .pipe(source('bundle.js'))
    .pipe(gulp.dest('dist'));
});

// Clean Task
gulp.task('clean', function (cb) {
  return del(['dist'], cb);
});

// Watch Task
gulp.task('watch', function() {
  gulp.watch('app/css/*.css',
    gulp.series('styles', browsersync.reload));
  gulp.watch('app/js/*.js',
    gulp.series('scripts', browsersync.reload));
  gulp.watch('app/img/*',
    gulp.series('images', browsersync.reload));
});
```

Line 83, Column 1 Spaces: 4 JavaScript

Another way is that we could use globs to select all of the files inside of the `dist` folder, but leave the `dist` folder itself intact:

```
gulp.task('clean', function (cb) {
  del(['dist/*'], cb);
});
```

We could also delete all of our files inside the `dist` folder except a specific file, which we will leave untouched. We can accomplish this by prefixing the file path with an exclamation point, which is the logical NOT operator.

```
gulp.task('clean', function (cb) {
  del(['dist/*', '!dist/site.css'], cb);
});
```

External configuration

As you create or expand your `gulpfile`, you may reach a point where you would prefer to separate your configuration into an additional file. This is a common issue that arises as users get more comfortable with gulp and wish to implement more control over how they configure their builds.

This can easily be done by creating an additional `config.json` file with each of the configuration options you would like to specify:

```
{
  "desktop": {
    "src": [
      "assets/desktop/css/*.css",
      "assets/desktop/js/*.js"
    ],
    "dest": "dist/desktop"
  },
  "mobile": {
    "src": [
      "assets/mobile/css/*.css",
      "assets/mobile/js/*.js"
    ],
    "dest": "dist/mobile"
  }
}
```

Then, we can include it in our `gulpfile` like all of our plugins and modules by using a `require` function:

```
var config = require('./config.json');
```

The only difference with this require function is that you must prepend it with `./` to tell node that this file will reside in the main project directory instead of the `node_modules` directory, where all of the other plugins and modules reside.

Now, you can use this config in a number of ways to pass along the data inside it. You could simply access the information directly in any of your tasks.

The following code illustrates the use of the `config.json` we created earlier in our styles task:

```
gulp.task('styles', function () {
    return gulp.src(config.desktop.src)
        .pipe(concat('site.css'))
        .pipe(myth())
        .pipe(gulp.dest(config.desktop.dest));
});
```

Alternatively, you could build an additional helper that can be used to reduce repetition, and then pass the helper to the tasks as needed:

```
function processCSS(cfg) {
    return gulp.src(cfg.src)
        .pipe(concat('site.css'))
        .pipe(myth())
        .pipe(gulp.dest(cfg.dest));
}

gulp.task('styles', function () {
    processCSS(config.desktop);
    processCSS(config.mobile);
});
```

Task dependencies

When creating tasks, you might encounter a scenario in which you will need to ensure that a series of tasks run in a specific order.

As mentioned in the earlier chapters, `gulp` uses a special method, `.series`, that allows us to specify the order in which our tasks need to be executed.

In fact, we have already implemented this method in *Chapter 4, Using Node.js Modules for Advanced Tasks*, to ensure that we only tell BrowserSync to reload our browsers once our core tasks have completed.

To further take advantage of the `.series` method, we can additionally use it to ensure that our newly created `clean` task must complete before any other tasks can run.

The code below illustrates how we can add the `clean` task as the first argument of the `.series` method to ensure that it is run first in the sequence:

```
// Watch Task
gulp.task('watch', function() {
    gulp.watch('app/css/*.css', gulp.series('clean', 'styles',
        browsersync.reload));
    gulp.watch('app/js/*.js', gulp.series('clean', 'scripts',
        browsersync.reload));
    gulp.watch('app/img/*', gulp.series('clean', 'images',
        browsersync.reload));
});
```

The next screenshot illustrates this change:

```
gulpfile.js -- gulp-book
FOLDERS
gulp-book
  app
  node_modules
  gulpfile.js
  index.html
  package.json
baseDir:'./'
}
}, cb);
});
// Browserify Task
gulp.task('browserify', function() {
    return browserify('./app/js/app.js')
        .bundle()
        .pipe(source('bundle.js'))
        .pipe(gulp.dest('dist'));
});
// Clean Task
gulp.task('clean', function (cb) {
    return del(['dist'], cb);
});
// Watch Task
gulp.task('watch', function() {
    gulp.watch('app/css/*.css',
        gulp.series('clean', 'styles', browsersync.reload));
    gulp.watch('app/js/*.js',
        gulp.series('clean', 'scripts', browsersync.reload));
    gulp.watch('app/img/*',
        gulp.series('clean', 'images', browsersync.reload));
});
```

3 selection regions Spaces: 4 JavaScript

Source maps

Minifying your JavaScript source code into distributable files can be a rough experience when it comes to debugging in the browser. Anytime you hit a snag and check your console for errors, it simply leads to the compiled and unreadable code.

Modern browsers have some features that will make their best attempt to make the compiled code readable; however, all of the variable and function names have likely been renamed to save on file size. This is still too unreadable to be practical and beneficial.

The solution to this is to generate source maps that will allow us to view the unbuilt versions of our code in the browser so that we can properly debug it.

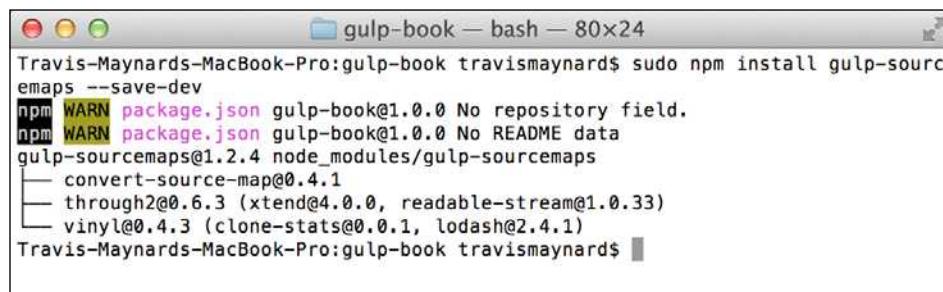
Since we have already established a `scripts` task, you can simply add an additional plugin called `gulp-sourcemaps` that you can introduce into our pipechain, which will generate those source maps for us.

Installing a source maps plugin

To begin, we must first install the `gulp-sourcemaps` plugin:

```
npm install gulp-sourcemaps --save-dev
```

The following screenshot displays the successful installation response:



```
gulp-book — bash — 80x24
Travis-Maynards-MacBook-Pro:gulp-book travismaynard$ sudo npm install gulp-sourcemaps --save-dev
npm WARN package.json gulp-book@1.0.0 No repository field.
npm WARN package.json gulp-book@1.0.0 No README data
gulp-sourcemaps@1.2.4 node_modules/gulp-sourcemaps
  └── convert-source-map@0.4.1
    ├── through2@0.6.3 (xtend@4.0.0, readable-stream@1.0.33)
    └── vinyl@0.4.3 (clone-stats@0.0.1, lodash@2.4.1)
Travis-Maynards-MacBook-Pro:gulp-book travismaynard$
```

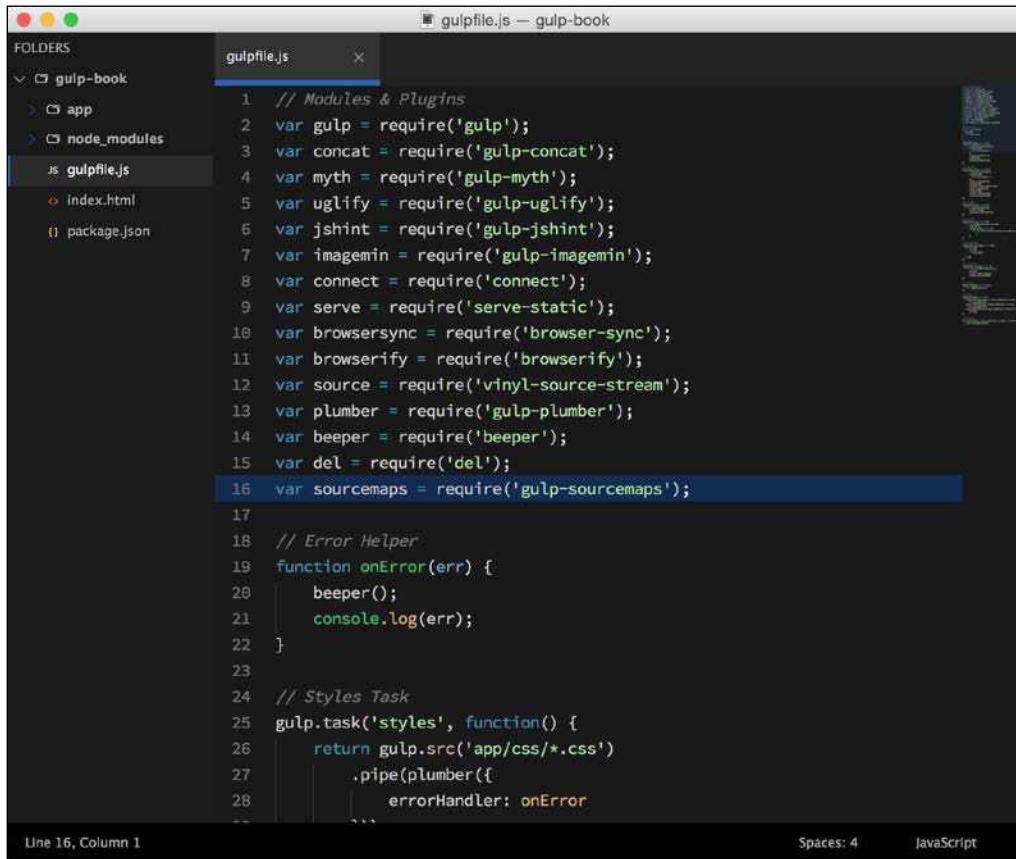
Including a source maps plugin

Once the plugin has been installed, we need to add it in our `gulpfile`:

```
// Modules & Plugins
var gulp = require('gulp');
var concat = require('gulp-concat');
var myth = require('gulp-myth');
```

```
var uglify = require('gulp-uglify');
var jshint = require('gulp-jshint');
var imagemin = require('gulp-imagemin');
var connect = require('connect');
var serve = require('serve-static');
var browsersync = require('browser-sync');
var browserify = require('browserify');
var source = require('vinyl-source-stream');
var plumber = require('gulp-plumber');
var beeper = require('beeper');
var del = require('del');
var sourcemaps = require('gulp-sourcemaps') // Added
```

The next screenshot reflects our required plugin:



```
gulpfile.js
```

```
1 // Modules & Plugins
2 var gulp = require('gulp');
3 var concat = require('gulp-concat');
4 var myth = require('gulp-myth');
5 var uglify = require('gulp-uglify');
6 var jshint = require('gulp-jshint');
7 var imagemin = require('gulp-imagemin');
8 var connect = require('connect');
9 var serve = require('serve-static');
10 var browsersync = require('browser-sync');
11 var browserify = require('browserify');
12 var source = require('vinyl-source-stream');
13 var plumber = require('gulp-plumber');
14 var beeper = require('beeper');
15 var del = require('del');
16 var sourcemaps = require('gulp-sourcemaps');

17
18 // Error Helper
19 function onError(err) {
20     beeper();
21     console.log(err);
22 }
23
24 // Styles Task
25 gulp.task('styles', function() {
26     return gulp.src('app/css/*.css')
27         .pipe(plumber({
28             errorHandler: onError
29         })
30     );
31 }

32
33 // Default Task
34 gulp.task('default', function() {
35     browserSync.reload();
36 });

37
38 // Watch Task
39 gulp.task('watch', function() {
40     browserSync.reload();
41 });

42
43 // Run Tasks
44 gulp.task('run', ['watch', 'styles']);
45
```

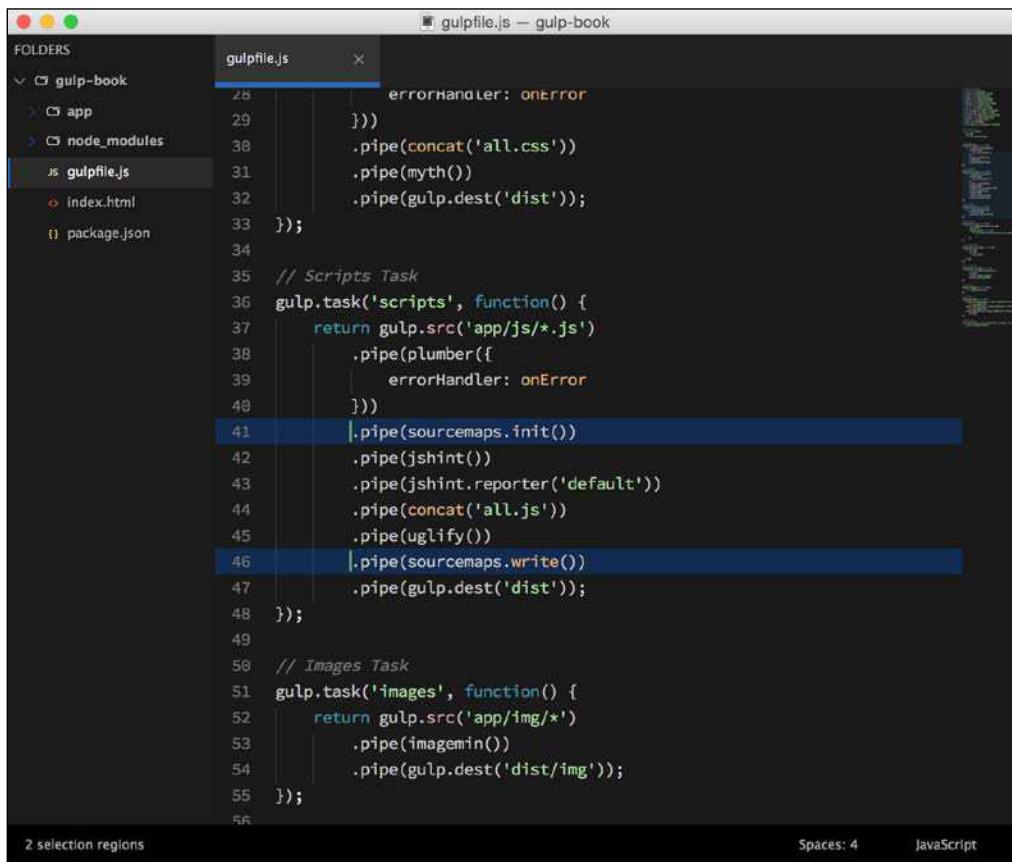
Line 16, Column 1 Spaces: 4 JavaScript

Adding source maps to the PipeChain task

Now that the plugin has been installed, you can jump back to the `scripts` task that you created in *Chapter 2, Getting Started* and fit the new plugin into the pipechain.

```
gulp.task('scripts', function() {
    return gulp.src('app/js/*.js')
        .pipe(sourcemaps.init()) // Added
        .pipe(concat('all.js'))
        .pipe(jshint())
        .pipe(jshint.reporter('default'))
        .pipe(uglify())
        .pipe(sourcemaps.write()) // Added
        .pipe(gulp.dest('dist'));
});
```

The following screenshot displays our modified pipechain:



The screenshot shows a code editor window with the file `gulpfile.js` open. The code defines two Gulp tasks: `scripts` and `images`. The `scripts` task uses the `sourcemaps` plugin to generate source maps for concatenated JavaScript files. The `images` task uses the `imagemin` plugin to compress image files. The code editor interface includes a sidebar for navigating through project folders like `gulp-book`, `app`, and `node_modules`.

```
gulpfile.js — gulp-book
FOLDERS
gulp-book
  app
  node_modules
  gulpfile.js
  Index.html
  package.json
  28   errorHandler: onerror
  29   }));
  30   .pipe(concat('all.css'))
  31   .pipe(myth())
  32   .pipe(gulp.dest('dist'));
  33 });
  34
  35 // Scripts Task
  36 gulp.task('scripts', function() {
  37   return gulp.src('app/js/*.js')
  38     .pipe(plumber({
  39       errorHandler: onError
  40     }));
  41     .pipe(sourcemaps.init())
  42     .pipe(jshint())
  43     .pipe(jshint.reporter('default'))
  44     .pipe(concat('all.js'))
  45     .pipe(uglify())
  46     .pipe(sourcemaps.write())
  47     .pipe(gulp.dest('dist'));
  48 });
  49
  50 // Images Task
  51 gulp.task('images', function() {
  52   return gulp.src('app/img/*')
  53     .pipe(imagemin())
  54     .pipe(gulp.dest('dist/img'));
  55 });

  56
```

2 selection regions Spaces: 4 JavaScript

In this code, we have added two lines. One has been added at the very beginning of the pipechain to initialize our source map plugin. The second has been added just before our pipe to gulp's `dest()` method. This code will save our source maps inline with our compiled JavaScript file.

You can also save the source map as an additional file if you would prefer to keep your compiled code and your source maps separate. Instead of executing the `.write()` method without any arguments, you can pass in a path to instruct it to save your source map into a separate file:

```
gulp.task('scripts', function() {
    return gulp.src('app/js/*.js')
        .pipe(sourcemaps.init()) // Added
        .pipe(concat('all.js'))
        .pipe(jshint())
        .pipe(jshint.reporter('default'))
        .pipe(uglify())
        .pipe(sourcemaps.write('dist/maps')) // Added
        .pipe(gulp.dest('dist'));
});
```

Summary

In this chapter we discussed valuable tips and tricks to help resolve some common issues that users can run into while using gulp.

By implementing some additional gulp plugins and node modules, we were able to make our tasks even more helpful and powerful.

We explored how to implement better error handling and prevented our watch task from silently exiting upon failure.

Using arrays and external configs, we were able to gain more control over how our source files are processed, and how to prevent unnecessary repetition throughout our tasks.

We discussed how to use tasks as dependencies and used them to implement a new task to clean our build directory upon task execution.

Finally, we discussed how to improve browser debugging by adding source map generation into our JavaScript task.

Key Features and Additional Resources

Before we wrap things up, let's take a moment to review the content that we have covered throughout the book and take a look at some key points to remember moving forward.

Chapter 1 – Introducing Gulp

In the first chapter, we focused primarily on understanding the languages and tools to be used throughout the book. You learned that gulp is a build tool built upon the node.js platform to perform automated tasks for your development projects.

In this chapter, we covered the following topics:

- Node.js and npm are the backbone behind gulp
- Gulp is great for automation, optimization, and efficiency
- Gulp uses streams to process files, which allows you to have an incredibly fast execution and fine control of how each file is processed

Chapter 2 – Getting Started

In this chapter, we focused on getting our local environment set up by installing any software that is needed to move forward. We learned how to use a command-line interface and the anatomy of a gulpfile.

In this chapter, we covered the following topics:

- Gulp's main methods are: `.task()`, `.src()`, `.dest()`, `.watch()`, `.parallel()`, and `.series()`.
- All tasks share a common structure and syntax
- The `sudo` command will help you resolve permissions errors

Chapter 3 – Performing Tasks with Gulp

In this chapter, we created a set of base tasks in which we used to build upon in the following chapters. These base tasks included concatenation, minification, and preprocessing of our source files.

In this chapter, we covered the following topics:

- Gulp tasks share a very common structure that is easy to learn and remember
- The `default` task is what Gulp executes when no task name is specified
- You can stop any ongoing command-line process, such as the `watch` task, by pressing `Ctrl + C`

Chapter 4 – Using Node.js Modules for Advanced Tasks

In this chapter, we explored how to use node.js modules instead of gulp plugins in our tasks. We created new tasks to run a static server, keep our project in sync across devices, and take advantage of node.js' module definitions in our client-side code.

In this chapter, we covered the following topics:

- If you can complete the task with a plain node.js module, then you likely should
- When developing gulp plugins, don't recreate something that has already been created. Instead, offer to help improve the current plugin
- The Gulp team actively blacklists duplicate plugins and plugins that act as wrappers around plain Node modules to keep the plugin ecosystem clean

Chapter 5 – Resolving Issues

In this chapter, we discussed ways to improve upon our tasks by improving error handling, ordering our source files, and cleaning up our compiled code. We learned how to set up task dependencies, generate source maps, and use an external configuration file.

In this chapter, we covered the following topics:

- Gulp's default error handling process will be much improved in future releases, but until then, the `gulp-plumber` plugin will give you the additional control you need
- The source order can be handled by introducing an ordered array of source file paths
- In your source arrays, you can use direct paths to files along with globs without having to worry about the direct file path being processed twice

References

The content of this book offers a great introduction to gulp, but there are many additional resources that are resourceful, as you continue to use and learn more about gulp. As you move forward, I highly suggest that you take a look at these resources to pick up on additional tips and examples that were outside the scope of this book.

Stream-handbook

If you're interested in a more thorough explanation of streams and how they work, then substack's stream-handbook is exactly what you're looking for. This is one of the best available resources for streams on the Internet. For more information, refer to <https://github.com/substack/stream-handbook>.

NodeSchool

NodeSchool is a fantastic command-line resource for learning many different types of languages and technologies. This is a great place to learn more about both node.js and streams; you can then put your newly acquired command-line skills into practice.

If you would like to learn more about node.js, be sure to check out the [learnyounode](#) course. If you're interested in learning more about streams, the [stream-adventure](#) course is just what you're looking for. For more information, refer to <http://nodeschool.io/>.

Gulp recipes

The gulp community keeps an updated list of "recipes" for common use cases in the official gulp documentation. These recipes are great to implement into projects or use as starting points for more advanced uses. For more information, refer to <https://github.com/gulpjs/gulp/tree/master/docs/recipes>.

Index

Symbols

.on() method 65
-v command 28

A

anatomy, gulpfile
dest() method 34
modules, including 35
pipe() method 34
plugins, including 35
src() method 34
task() method 33
watch() method 34

B

beeper
including 81
Browserify
about 71
modules, including 72
modules, installing 71
task, writing 73-75
URL 75
BrowserSync
about 66
including 67
installing 66, 67
task, writing 68-70

C

cd command 15
clean task
writing 86, 87
command-line
about 11
Mac OS X terminal 12
Windows PowerShell 12
working with 11-14
command reference
cd command 15
ls command 14
mkdir command 16
ni command 17
sudo keyword 18
touch command 17
content
adding, to project 24
CSS
preparing 25

D

default task
about 57
gulpfile 58
running 59
writing 57
del module
including 85
installing 85

dest() method 34

directory

 changing 15, 16

 creating 16

E

error helper function

 writing 82, 83

errors, handling

 about 77

 beeper, including 81

 beeper, installing 80

 error helper function, writing 82, 83

 gulp-plumber, including 80

 gulp-plumber, installing 78

external configuration, gulpfile 87, 88

F

file

 creating, on Mac/Linux 17

 creating, on Windows 17

 listing 14

folder

 changing 15, 16

 creating 16

 listing 14

G

gulp

 about 6, 95

 build file, writing in code 9

 global installation 33

 installing 30

 key points 95

 local installation 31, 32

 locating 30

 need for 7-9

 overview 5, 6

 project automation 7, 8

 recipes, URL 98

 streams 9

gulpfile

 anatomy 33

 creating 37

gulp-plumber

 including 79, 80

 installing 78

H

HTML file

 preparing 24

I

images

 adding 27

images task

 about 50

 gulp plugins, including 52

 gulp plugins, installing 50, 51

 reviewing 54

 writing 53

issues

 key points 97

 resolving 97

J

JavaScript

 preparing 26

L

Less 40

ls command 14

M

Mac/Linux

 file, creating 17

mkdir command 16

modules, Browserify

 installing 71

modules, static server

 including 63

 installing 62

Myth 40

N

ni command 17
node.js
 about 7
 installation, verifying 28
 installing 27
 URL 27
node modules
 key points 96
 using, for advanced tasks 96
NodeSchool
 about 97
 URL 98
npm
 downloading 27
 installation, verifying 28
 installing 27
 URL 30

P

package.json file
 creating 29
PipeChain task
 source maps, adding 92, 93
pipe() method 34
plain node.js modules
 need for 61, 62
plugin, gulp
 using 39, 40
 using, with default task 57
 using, with images task 50
 using, with scripts task 45
 using, with styles task 40
 using, with watch task 54
project
 content, adding 24
 structure, creating 18-24
project cleanup
 about 85
 del module, including 85
 del module, installing 85

R

references
 gulp recipes 98
 NodeSchool 98
 stream-handbook 97
require() function 71

S

Sass 40
scripts task
 about 45
 gulp plugins, including 47
 gulp plugins, installing 45, 46
 writing 47-49
server task
 writing 64-66
single task
 running 59
source maps
 about 90
 adding, to PipeChain task 92, 93
 plugin, including 90, 91
 plugin, installing 90
source ordering 84
src() method 34
static server
 about 62
 modules, including 63
 modules, installing 62
 server task, writing 64-66
stream-handbook
 URL 97
streams 9
styles task
 gulp plugins, including 42
 gulp plugins, installing 40, 41
 preprocessors 44, 45
 reviewing 45
 writing 42-44
sudo keyword 18

T

task
default task, running 59
single task, running 59
writing 36
task() method 33
touch command 17

W

watch() method 34
watch task
about 54
reviewing 57
stopping 60
writing 54-56
Windows
file, creating 17



Thank you for buying Getting Started with Gulp

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.

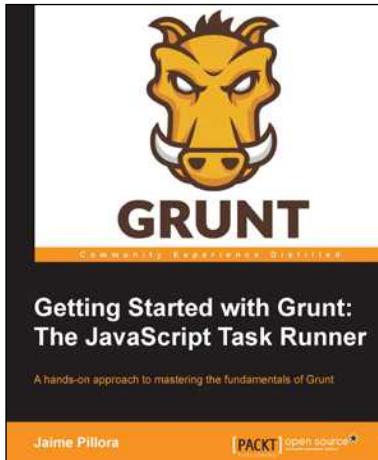
About Packt Open Source

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.

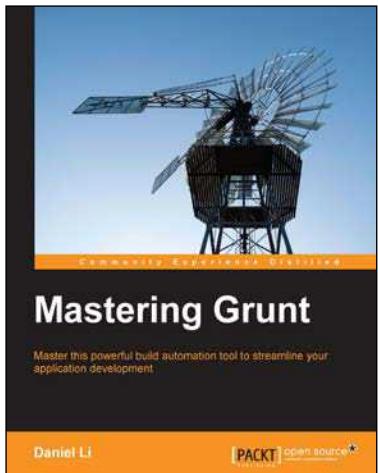


Getting Started with Grunt: The JavaScript Task Runner

ISBN: 978-1-78398-062-8 Paperback: 132 pages

A hands-on approach to mastering the fundamentals of Grunt

1. Gain insight on the core concepts of Grunt, Node.js and npm to get started with Grunt.
2. Learn how to install, configure, run, and customize Grunt.
3. Example-driven and filled with tips to help you create custom Grunt tasks.



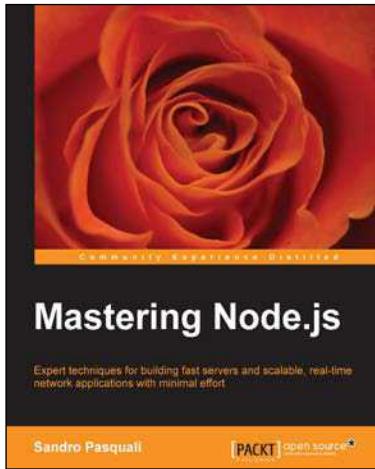
Mastering Grunt

ISBN: 978-1-78398-092-5 Paperback: 110 pages

Master this powerful build automation tool to streamline your application development

1. Master the development of your web applications by combining Grunt with an army of other useful tools.
2. Learn about the key tasks behind devops integration and automation so you can utilize Grunt in a team-working environment.
3. Accelerate your web development abilities by employing best practices, including SEO, page speed optimization, and responsive design.

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

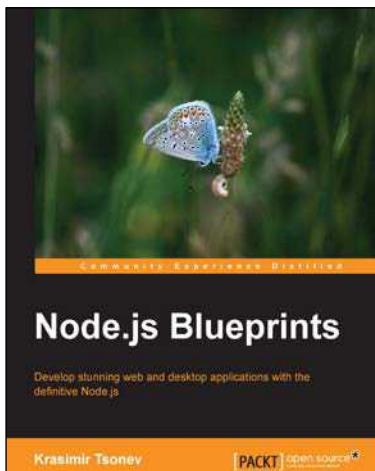


Mastering Node.js

ISBN: 978-1-78216-632-0 Paperback: 346 pages

Expert techniques for building fast servers and scalable, real-time network applications with minimal effort

1. Master the latest techniques for building real-time, big data applications, integrating Facebook, Twitter, and other network services.
2. Tame asynchronous programming, the event loop, and parallel data processing.
3. Use the Express and Path frameworks to speed up development and deliver scalable, higher quality software more quickly.



Node.js Blueprints

ISBN: 978-1-78328-733-8 Paperback: 268 pages

Develop stunning web and desktop applications with the definitive Node.js

1. Utilize libraries and frameworks to develop real-world applications using Node.js.
2. Explore Node.js compatibility with AngularJS, Socket.io, BackboneJS, EmberJS, and GruntJS.
3. Step-by-step tutorials that will help you to utilize the enormous capabilities of Node.js.

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