



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

Mastering Grunt

Master this powerful build automation tool to streamline your application development

Daniel Li

[PACKT] open source*
PUBLISHING community experience distilled

www.it-ebooks.info

Mastering Grunt

Master this powerful build automation tool
to streamline your application development

Daniel Li



BIRMINGHAM - MUMBAI

Mastering Grunt

Copyright © 2014 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: April 2014

Production Reference: 1180414

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

ISBN 978-1-78398-092-5

www.packtpub.com

Cover Image by Michael Harms (kunstraum@googlemail.com)

Credits

Author

Daniel Li

Project Coordinator

Harshal Ved

Reviewers

Florian Bruniaux

Philippe Charrière

Peter deHaan

Proofreaders

Maria Gould

Bernadette Watkins

Commissioning Editor

Kartikey Pandey

Indexer

Monica Ajmera Mehta

Acquisition Editor

Richard Harvey

Graphics

Ronak Dhruv

Content Development Editor

Anila Vincent

Production Coordinators

Pooja Chiplunkar

Manu Joseph

Technical Editors

Manan Badani

Indrajit Das

Cover Work

Pooja Chiplunkar

Copy Editors

Mradula Hegde

Gladson Monteiro

Deepa Nambiar

Kirti Pai

Alfida Paiva

About the Author

Daniel Li is currently an independent consultant for small- and medium-sized businesses, and resides in Waterloo, Ontario. Having gained experience at over a dozen institutions since 2009, he leverages his knowledge of Grunt.js and modern web development in writing this book. He has won over \$20,000 in coding competitions since 2009, and most recently won the Kik Cup Hackathon in Fall 2013. His open source contributions over the last three years helped him earn a place as a finalist in Canada's Top 20 Under 20 2013 list. He occasionally answers questions on the collaborative question and answer website, stackoverflow.com, as a top 4 percent user. He has also authored *Instant Brainshark*, Packt Publishing.

I would like to dedicate this book to all those who believed in me.

About the Reviewers

Florian Bruniaux is a French student at the University of Technology of Troyes (UTT), and is studying in the IT and Information Systems department. He is passionate about new technologies, particularly of process optimization and software development.

Specialized in frontend and client-side development, he has worked for various companies such as Aylan, a French start-up, Oxyane, and EDF, where he worked on IT projects such as the server-monitoring system, cross-browser, multidevice app conception, and development.

I would like to thank Steve Burghgraeve, an IT engineer at Oxyane, and Aurélien Bénél, a teacher, researcher, and lecturer in Computer Science at UTT, for their help in my different projects and all the knowledge they've transferred to me.

Philippe Charrière is a bid manager at Steria in France, and at night, he is an open source developer advocate for the Golo project (<http://golo-lang.org/>) and is a Backbone enthusiast. He has written a small French open source book about Backbone.js (<https://github.com/k33g/backbone.en.douceur/>). He is also an occasional speaker on Backbone.js and mobile technologies. He focuses primarily on open web technologies (frontend and server-side).

Peter deHaan likes Grunt a lot and thinks that it's the best thing to happen to Node.js since npm. You can follow his Grunt npm-Twitter-bot feed using the handle `@gruntweekly`.

www.PacktPub.com

Support files, eBooks, discount offers, and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

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.



<http://PacktLib.PacktPub.com>

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

Table of Contents

Preface	1
Chapter 1: Getting Started	5
Introducing Grunt	5
Plugins and Gruntfile.js	6
Dissecting the Gruntfile.js configuration file	7
Constants and functions	7
Configuration	7
User-defined tasks	8
Introducing Git	8
Using Git	9
Installing Git	9
Installing Git on Windows	9
Installing Git on Mac OS X	9
Installing Git on Linux	9
Git basics	10
Introducing GitHub	11
Using GitHub	11
Installing GitHub	12
Introducing npm	13
Why use npm?	14
Installing npm	14
Installing npm on Windows	14
Installing npm on Mac OS X and Linux	14
Using npm	15
Introducing Bower	16
Using Bower	16
Installing Bower	16
Bower basics	18

Installing Grunt	18
Installation steps	19
Troubleshooting	19
Grunt: command not found	19
Fatal error: Unable to find local grunt	19
Deploying a Hello World page	20
Summary	24
Chapter 2: Developing a Blog with Jade and Sass	25
A brief summary of Jade	25
A brief summary of Sass	26
What is Sass?	26
Concatenation and minification	27
Building the blog	27
Installing the required Grunt plugins	27
Configuring grunt-contrib-watch	28
Developing the blog	32
Implementing the custom build Grunt task	40
Summary	43
Chapter 3: Making an Employee Management System	45
A brief summary of CoffeeScript	45
Building the employee management system	46
Installing the required Grunt plugins	46
Configuring grunt-contrib-watch	47
Developing the employee management system	49
Implementing the custom build Grunt task	56
Summary	59
Chapter 4: Final Project – Simple Bulletin Board System	61
Installing the required Grunt plugins	62
Configuring grunt-contrib-watch	63
Developing a simple Bulletin Board System (BBS)	65
Writing Mocha tests using Zombie and Assert	68
Implementing the custom test Grunt task	70
Implementing the custom build Grunt task	71
Summary	74

Chapter 5: Best Practices for Modern Web Applications	75
The importance of search engine optimization	75
Item 1 – using keywords effectively	75
Item 2 – header tags are powerful	76
Item 3 – make sure to have alternative attributes for images	76
Item 4 – enforcing clean URLs	77
Item 5 – backlink whenever safe and possible	78
Item 6 – handling HTTP status codes properly	78
Item 7 – making use of your robots.txt and site map files	79
Using Grunt to reinforce SEO practices	80
Form validation in the modern web world	81
Item 8 – using client-side validation over error pages	81
Item 9 – differentiating required and optional information	82
Item 10 – avoiding confusing fields	82
Item 11 – using confirmation fields for pertinent data	83
Item 12 – using custom inputs for complex data types	83
Item 13 – preventing autovalidation with CAPTCHAs	84
Item 14 – reinforcing data integrity with server-side validation	85
Using Grunt to automate form testing	85
Designing interfaces for the mobile generation	85
Item 15 – designing preemptively with mobile in mind	86
Item 16 – lazy load content using JavaScript	87
Item 17 – defer parsing of JavaScript	88
Using Grunt to reduce page load time	88
Summary	89
Index	91

Preface

Grunt.js is primarily used for DevOps integration. Being able to automate compression, conversion, and obfuscation, developers and sysadmins are able to deploy projects in a fast and easy way. Previous solutions have required too much knowledge. With the ease of using Grunt plugins and configuration files, it allows developers to work along with sysadmins throughout the integration process.

What this book covers

Chapter 1, Getting Started, gives a brief introduction to Grunt.js for readers. It will introduce the basic concepts required to understand how Grunt.js works and why automated integration is important. The project in this chapter will set up users with all the tools required for the upcoming projects from this point forward. It covers the installation of software dependencies including Git, Bower, and Grunt.js.

Chapter 2, Developing a Blog with Jade and Sass, will look into Jade, a templating engine originally developed for the Node.js platform. It will involve developing a blog as a use case, using templates for individual posts and the blog as a whole. The project will also emphasize the importance of compression, minification, and obfuscation in developing a high-traffic blog as a use case.

Chapter 3, Making an Employee Management System, will look into CoffeeScript, a language that compiles to JavaScript. It will involve developing an employee management system as a use case, demonstrating CoffeeScript's easy-to-use classes and coding practices.

Chapter 4, Final Project – Simple Bulletin Board System, will be the largest project in the book. It will involve creating a simple BBS website, also known as a message board or forum, using all the concepts involved in this book. We will illustrate the importance of using test-driven development via Mocha throughout this chapter.

Chapter 5, Best Practices for Modern Web Applications, will cover the best practices that are used today for frontend development. It will cover, search engine optimization, form validation, user experience/interface design, and responsive design.

What you need for this book

In order to complete the projects within this book, you will need to have the following available beforehand. A tutorial on how to install and configure the various software dependencies may be found in *Chapter 1, Getting Started*.

- A machine that has its default terminal shell set to bash will be required. Mac OS X Version 10.3 or higher will complete this requirement. Most popular distributions of Linux will also come with bash as its default. If your primary operating system is Windows, you can use your existing command line for this book.
- Git will be required for downloading the project templates from GitHub, an online open source project-hosting service. GitHub will also provide a separate set of online installation instructions for each project on its README page.
- npm will be used for installing a variety of dependencies including Grunt.js and its plugins. Node.js and server-side concepts will not be covered in this book.
- Bower is a package management system, which will be used for client-side, frontend JavaScript libraries in this book. It will depend on npm for installation.
- Lastly, Grunt.js will be required for all projects in this book. Intense emphasis will be put on how to properly configure Grunt.js along with plugins. Grunt.js will depend on npm for installation.

Who this book is for

This book is designed for professional developers and sysadmins who would like an in-depth learning approach to Grunt.js to ensure that their projects are optimally configured. Hobbyist developers are also encouraged to go through the chapters to broaden their horizons on various web tools. Lastly, managers and entrepreneurs may be interested in an overview of Grunt.js and its importance when the tool has been chosen for a particular project.

Conventions

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

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "Open the `index.html` file in the `src` folder in a web browser of your choice."

A block of code is set as follows:

```
// constants and functions


module.exports = function (grunt) {
  grunt.initConfig({
    // configuration
  });


  // user-defined tasks
}
```

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

```
# npm install -g bower
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Fill out the necessary fields on the homepage and click on the **Sign Up for GitHub** button."

[ Warnings or important notes appear in a box like this.]

[ Tips and tricks appear like this.]

Reader feedback

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

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

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

Customer support

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

Errata

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

Piracy

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

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

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

Questions

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

1

Getting Started

Over the recent years, the open source community has come up with great tools that have eased development, such as CSS preprocessors, simple-to-use testing libraries, and minification/compression libraries. Aimed to facilitate the automation of tasks, Grunt helps both developers and sysadmins by speeding up the development time and easing the production integration process.

This chapter will reintroduce you to Grunt and demonstrate the importance of various other software tools. As you go further into the book, you will learn how to develop your own projects using Grunt, with an increase in complexity and applicability. In particular, you will acquire knowledge of how to use Grunt efficiently as a software integration tool as well as learning about the best practices surrounding web development today.

Introducing Grunt

Grunt is a simple-to-use task runner written with Node.js, a scalable JavaScript software platform. A task runner is defined as a software tool used to automate predefined tasks to ease the development and integration of large-scale projects that are in production.

The following is a look into the official website for Grunt:



Source: <http://gruntjs.com/>

Grunt is currently used by large organizations such as Twitter and Adobe, and by technologies such as jQuery. Through consistent updates, it has recently become one of the most stable automation tools since its creation. The open source community has embraced its simplicity compared to alternatives such as Ant or Maven. Lastly, it is praised for its use of the ubiquitous language, JavaScript, thereby easing the training among development teams. In this book, Grunt will be used to automate the compilation of Sass to CSS and CoffeeScript to JavaScript. Minification will be used to optimize page load times. Obfuscation, the mangling of information, will also be automated to deter code theft and manipulation from the client side. Grunt will be used to automate testing via the Mocha engine.

So, why should you use Grunt? In today's world of modern web applications, you may find yourself using various tools that require compilation or preprocessing. Likewise, you may wish to obfuscate code and minimize the size of your public files in order to optimize the load time of our website. At other times, you could manually call executables to perform these tasks. This would only waste more time that would otherwise have been spent in the development of your app. With a simple, customizable configuration file, Grunt allows you to set up a build script to automate these activities via its community-curated plugins.

Plugins and Gruntfile.js

Plugins are installed via **npm**, which will be explained later in this chapter. Certain plugins are deemed **contrib** packages (these are labeled with a contrib indicator such as `grunt-contrib-jshint`) and are branded as officially maintained and stable. These will be used throughout the book to ensure consistency.

Dissecting the Gruntfile.js configuration file

Every Gruntfile will be typically aligned with the following format:

```
// constants and functions

module.exports = function (grunt) {
  grunt.initConfig({
    // configuration
  });

  // user-defined tasks
}
```



It is important to note that Grunt follows the **CommonJS** spec, a project developed to normalize JavaScript styles and conventions. As such, Grunt exports itself as a module that contains its configuration and tasks.

Constants and functions

In order to ease the configuration process, Grunt users should ideally store any ports, functions, and other constants, which are used at the top of the file, as global variables. This ensures that if a constant or function suddenly changes, editing the global variable at the top would be a lot simpler than changing its value at every location in the file.

Also, constants help to provide information by allocating a variable name to each unknown value. For instance, look at the following command line:

```
var LIVERELOAD_PORT = 35729;
```

In this case, any instance of the port within the configuration can easily be traced back to the LiveReload plugin.

Configuration

The configuration section contains the settings required for each Grunt plugin that is included. As a result, these will be plugin-specific and will be documented by the maintainers of each plugin respectively.

For instance, the `grunt-contrib-uglify` plugin is used to minify various files. By default, the plugin will also obfuscate the existing code. Users can optionally turn the `mangle` option off in the configuration section to prevent this from happening.

User-defined tasks

Lastly, the Gruntfile will end with a list of user-defined tasks. By default, every Grunt plugin has a respective task that may be called to achieve its desired output. User-defined tasks may be defined at the end of the Gruntfile to synchronously chain multiple Grunt plugin tasks together.

For instance, you may wish to define a task for sysadmins that will deploy a production-ready version of your web application by combining minification, concatenation, and compilation tasks. You may also wish to define a watch task for web developers to auto-compile CoffeeScript or Sass files that require compilation to be used. For a team of interns, a task that combines validation plugins for various coding languages may be used to prevent errors and aid their learning process.

Introducing Git

Git is a version control and source code management system developed for the Linux kernel. Its speed, team-based flexibility, and related open source tools have led to Git's ubiquity in recent years. It supports the use of source code branching, revision histories, and bug tracking.

This book will use Git to pull in the base templates of projects used throughout this book. The steps that are required will be illustrated in subsequent chapters.

The following is a look into the official website for Git:



Source: <http://git-scm.com/>

Using Git

Let's suppose that you have a large-scale project developed by several different programmers. Over time, you will need a feasible way to collaborate on the project without overriding each other's changes. You'll also need to be sure to keep a revision history in order to have the ability to optionally revert the changes when necessary. This way, you'll also be able to track where a bug in your code is, depending on when it was first found, by using the revision history. These are all use cases for which Git comes in handy.

Installing Git

There are various ways to install and run Git. The following steps will illustrate how to exactly do this on various operating systems.

Installing Git on Windows

Perform the following steps to install Git on Windows:

1. Visit <http://git-scm.com/downloads> and click on the Windows link.
2. Run the `.exe` file that was just downloaded.
3. Follow the installation steps, and Git will be installed as a command-line utility alongside an optional GUI interface.

Installing Git on Mac OS X

Perform the following steps to install Git on Mac OS X:

1. Visit <http://git-scm.com/downloads> and click on the Mac OS X link.
2. Run the `.dmg` file that was just downloaded.
3. Go through the graphical installer process and you will end up with Git installed as a command-line utility.

Installing Git on Linux

Perform the following steps to install Git on Linux:

1. Visit <http://git-scm.com/downloads> and click on the Linux link.
2. Follow the instructions on the page for your respective Linux distribution. After issuing the necessary commands in a terminal, you will end up with Git installed as a command-line utility.

Git basics

In order to initialize a directory such that it may be used alongside Git, one must issue the following command:

```
git init
```

Prior to making a revision, one must add the files they have changed to a commit. To do this, you would issue the following command:

```
git add <file>
```

For example, issue the following command:

```
git add main.js
```

Once all the changed files have been added to the current revision, you are able to issue a commit, as shown in the following command:

```
git commit -m <commit message>
```

For instance, issue the following command:

```
git commit -m "Initial import of main.js file"
```

The current state of a Git-initialized directory is important. Knowing what files have been added to the current commit at any given time can help to prevent a premature commit. To query the current state, one would have to issue the following command:

```
git status
```

It also helps to reveal the current branch being used for development, a concept that will not be explored in this book.

Next, you will look into how to download a public Git repository, which can be done by inputting the following:

```
git clone <Git URL>
```

An example is shown as follows:

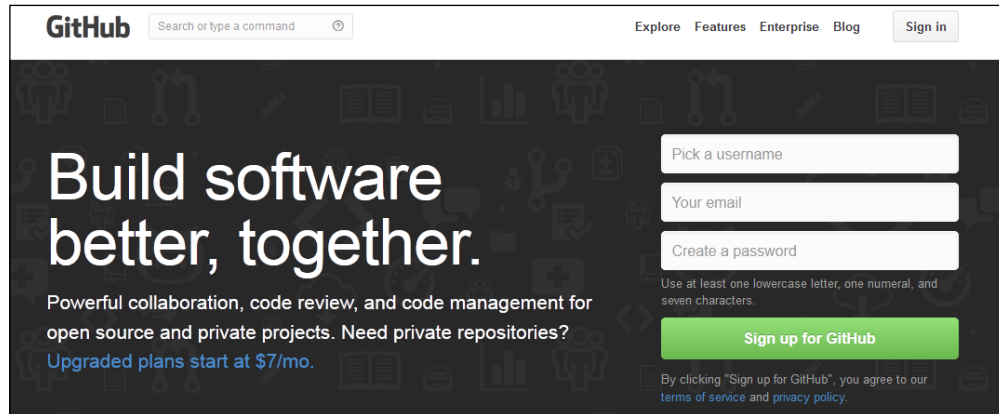
```
git clone https://github.com/packt-mg/Chapter-2.git
```

This command will be used throughout the book in order to download the necessary base templates for development.

Introducing GitHub

GitHub is an online hosting service primarily used for software projects. Using the version control system, Git users are able to deploy their coding projects to either public or private repositories on GitHub.

The following is a look into the official website for GitHub:



Source: <https://github.com/>

GitHub will be used in the book as a host to the base templates alongside extra documentation for the projects in this book. Readers should note that the main purpose of setting up GitHub is to have the ability to fork repositories. Forking allows one to clone a repository on the GitHub server, which allows one to keep track of their changes through commits and back up all of their code online in case of data loss.

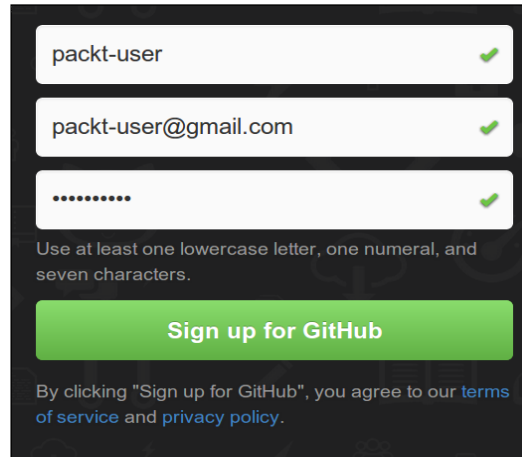
Using GitHub

By facilitating software collaboration, GitHub users can leverage its enterprise solutions to track development history and bugs, and contribute to an internal wiki. Using Git's branching capabilities and history tracking, it creates tree-based visual representations of software contributions for the team. These features, alongside the free hosting and backup of open source software, help to make GitHub the go-to source for software hosting.

Installing GitHub

The following steps will help to demonstrate how to sign up and use GitHub:

1. Go to <https://github.com/>.
2. Fill out the necessary fields on the home page and click on the **Sign Up for GitHub** button, as shown in the following screenshot:

A screenshot of the GitHub registration form. It features three input fields: a username field containing 'packt-user', an email field containing 'packt-user@gmail.com', and a password field with masked characters. Each field has a green checkmark icon to its right. Below the password field, there is a text instruction: 'Use at least one lowercase letter, one numeral, and seven characters.' A prominent green button labeled 'Sign up for GitHub' is positioned below the instructions. At the bottom, a line of text states: 'By clicking "Sign up for GitHub", you agree to our [terms of service](#) and [privacy policy](#).'

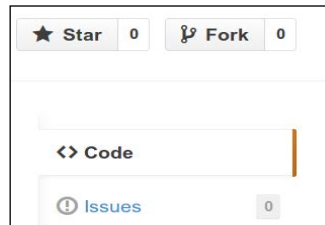
The GitHub registration form

3. Complete the installation steps as required.
4. Following this, visit the Packt Publishing Mastering Grunt GitHub repository at <https://github.com/packt-mg>.



The Packt Publishing GitHub page

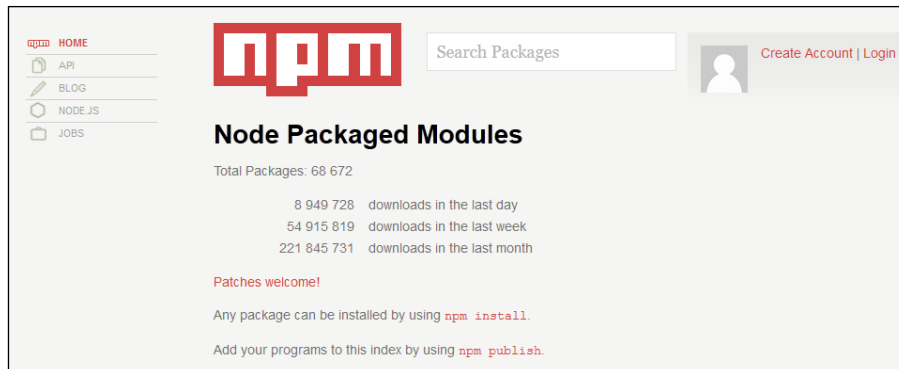
5. For each of the repositories, click on the **Fork** button in the top-right corner, as shown in the following screenshot, in order to clone it to your own account on the server side:



Introducing npm

npm is a package manager for the Node.js software platform. A package management system helps to streamline the installation, updating, and configuration of software packages. This way, one can easily manage the dependencies of a project, without any worries, in a scriptable manner.

The following is a look into the official website for npm:



Source: <https://www.npmjs.org/>

As Grunt is a JavaScript project built using Node, it naturally uses npm for version control. npm will also be used to install and update Grunt plugins and Bower. The npm registry also hosts a variety of other tools, including the Mocha testing engine, the Jade templating engine, and the ExpressJS web framework.

Why use npm?

Traditionally, if someone wanted to install a package that was built using Node.js, one would need to download the necessary files and build the package manually. Using npm, one can install and update software packages with a simple command. It also allows the installation of projects of a specific version, a technique that will be used throughout this book to ensure that project instructions align consistently with the software.

Installing npm

Since npm comes with Node, it suffices to install the Node platform to set up npm.

Installing npm on Windows

The following steps will cover the installation of npm on Windows through the Node installer:

1. Download the `.msi` installer located at <http://nodejs.org/download/>.
2. Run the `.msi` installer to install Node and npm.

Installing npm on Mac OS X and Linux

The following steps will cover the installation of npm via the command line on Mac OS X and Linux:

1. Download the `.pkg` installer located at <http://nodejs.org/download/>.
2. Run the `.pkg` installer to install Node and npm.
3. Optionally, if you are using Linux, determine your Linux distribution.
4. Once you have found this, install Node for your specific distribution or Mac OS X, by following the instructions mentioned in the link <https://github.com/joyent/node/wiki/Installing-Node.js-via-package-manager>.

Once Node is installed, you can test npm by calling the following binary:

`npm`

You should see a trace similar to the following:

Usage: npm <command>

where <command> is one of:

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

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

Using npm

To install a package in your current directory, you'll simply need to issue the following command:

npm install <package>

For instance:

npm install grunt

Note that at certain times, you may be interested in installing a package globally. To do this, you'll need to pass the `-g` flag, as shown in the following command:

npm install -g <package>

You may be interested in the Node packages installed in your current directory. You can list them using the following command:

npm ls

To update all the packages in your current directory, issue a simple update command. This is shown as follows:

npm update

Introducing Bower

Bower is a package manager for frontend web development. It will be used to install client-side libraries for the various projects in this book. Bower is similar to npm in that it helps to facilitate the installation, upgrading, and configuration of frontend web packages. The main difference is in the underlying implementation of Bower and npm. While npm uses a nested dependency tree and may involve the installation of multiple versions of a package, Bower keeps its dependency tree flat. This indicates that each client that uses Bower should only need one version per dependency installed, thereby alleviating the need for extra space.

The following is a look into the official website for Bower:



Source: <http://bower.io/>

Using Bower

Without Bower, one would traditionally download frontend libraries from their official websites and place them in their projects. Unfortunately, these libraries will not include the metadata that is necessary to automate the upgrading or configuration of the libraries. Using Bower and its internal cache, one is able to easily update, install, or remove packages on demand.

Installing Bower

Follow the given steps to install Bower:

1. Once npm is installed, you can install Bower globally using the following command:

```
sudo npm install -g bower@1.2.8
```

The `-g` flag allows you to run Bower as a global binary. As long as you are connected to the Internet and have the necessary space for installation, you should not encounter any errors throughout this process. Version 1.2.8 of Bower will be used in this book to ensure consistency.

2. You can test Bower by calling the following binary in your command line:

`bower`

This should return a trace similar to the following:

Usage:

```
bower <command> [<args>] [<options>]
```

Commands:

<code>cache</code>	Manage bower cache
<code>help</code>	Display help information about Bower
<code>home</code>	Opens a package homepage into your favorite browser
<code>info</code>	Info of a particular package
<code>init</code>	Interactively create a bower.json file
<code>install</code>	Install a package locally
<code>link</code>	Symlink a package folder
<code>list</code>	List local packages
<code>lookup</code>	Look up a package URL by name
<code>prune</code>	Removes local extraneous packages
<code>register</code>	Register a package
<code>search</code>	Search for a package by name
<code>update</code>	Update a local package
<code>uninstall</code>	Remove a local package

Options:

<code>-f, --force</code>	Makes various commands more forceful
<code>-j, --json</code>	Output consumable JSON
<code>-l, --log-level</code>	What level of logs to report
<code>-o, --offline</code>	Do not hit the network
<code>-q, --quiet</code>	Only output important information
<code>-s, --silent</code>	Do not output anything, besides errors
<code>-V, --verbose</code>	Makes output more verbose
<code>--allow-root</code>	Allows running commands as root

See 'bower help <command>' for more information on a specific command.

Bower basics

In order to install a package, issue the following command:

```
bower install <package>
```

For instance:

```
bower install jquery
```

This command will install jQuery from its GitHub repository to your directory. If you're aiming to install a package at a specific version, you can specify it as follows:

```
bower install <package>#<version>
```

An example of this is as follows:

```
bower install jquery#1.9.1
```

This will install jQuery Version 1.9.1.

In case you have installed a package by accident, you may uninstall it, as shown in the following command:

```
bower uninstall <package>
```

For example, issue the following command:

```
bower uninstall modernizr
```

For the sake of this book, a Bower configuration (in the form of `bower.json`) will be included. This allows you to install all the required packages by running the following command:

```
bower install
```

In order to regenerate a `bower.json` file of your own, you can issue an initialization command, shown as follows:

```
bower init
```

Installing Grunt

In this section, the installation of Grunt and any potential setup issues will be addressed.

Installation steps

There are no use cases for which Grunt should be installed globally. The configuration of Grunt and its associated plugins will be provided in each of the projects within this book.

When starting a new project, you can install Grunt within the project directory by issuing the following command:

```
npm install grunt
```

Any Grunt plugins that will be used in the projects of this book will also be installed through npm. The installation steps for each plugin will be included in each project.

Troubleshooting

In this section, various troubleshooting errors will be explored and diagnosed.

Grunt: command not found

This error is typically caused by the lack of `grunt-cli`, the command-line interface for Grunt. As per the best practices suggested by the Node community itself, Grunt is intended to be installed locally for each project. Any command-line tools such as `grunt-cli` are intended to be installed globally. Thus, you should issue a global installation of `grunt-cli` in order to fix this problem, as shown in the following command:

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

Fatal error: Unable to find local grunt

When you receive this error, it is because a local Grunt binary cannot be found within the Node modules of the project. This typically occurs when `grunt-cli` is installed but Grunt is uninstalled or missing.

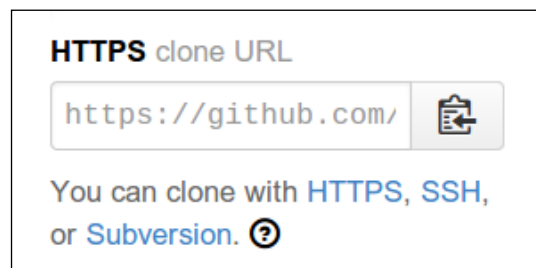
To fix this problem, just reinstall Grunt locally, ensuring that it is added to your `package.json` file by passing the `--save-dev` flag, as shown in the following command:

```
npm install --save-dev grunt
```


Deploying a Hello World page

We will now start the first project, which will involve the basics of Grunt, as shown in the following steps:

1. Visit <https://github.com/packt-mg/Chapter-1> for the project template.
2. In a terminal, traverse to a directory of your choice that will be used to store the project and its contents.
3. In your web browser, copy the HTTPS clone URL of the project. It can be found in a textbox similar to the one in the following screenshot:



4. Copy the HTTPS clone URL from the input box.
5. Issue a Git clone command by pasting the HTTPS clone URL in the terminal, as shown in the following command:

```
git clone <HTTPS clone url>
```
6. Traverse into the recently cloned directory by issuing the following command:

```
cd Chapter-1
```
7. Install the Node dependencies as follows:

```
npm install
```
8. You should then install the Bower dependencies as follows:

```
bower install
```
9. Open the `index.html` file in the `src` folder in a web browser of your choice. It should yield a page that outputs **Hello World**. By viewing the page source, it is evident that the page uses jQuery. We will make simple modifications to the Gruntfile to minimize and concatenate the JavaScript files.
10. Open `Gruntfile.js` in the root directory with a text editor of your choice.

11. You will be deploying the application to a folder named `dist`. Thus, we will need to clean the folder prior to use. Add the following object after the declaration of `config`:

```
var config = {};

config['clean'] = {
  build: {
    files: [{
      dot: true,
      src: [
        'dist/*',
        '!dist/.git*'
      ]
    }]
  }
};
```

This sets up the custom settings required for the `grunt-contrib-clean` plugin.



Note that the `dot` option, which is set as `true`, implies that the plugin will clean hidden files (prepended with a period) on Unix systems. Additionally, the `src` array specifies all the directories to be cleaned. The `!dist/.git` option prevents the Git metadata from being wiped out in case a subrepository has been created.

12. Next, you should clean the HTML file `index.html`. To do this, add the following object anywhere after the declaration of the `config` object:

```
config['htmlmin'] = {
  build: {
    options: {
      collapseBooleanAttributes: true,
      removeAttributeQuotes: true,
      removeRedundantAttributes: true,
      removeEmptyAttributes: true
    },
    files: [{
      expand: true,
      cwd: 'src',
      src: '{,*/}*.html',
      dest: 'dist'
    }]
  }
};
```

Note that the `collapseBooleanAttributes` option will aid in removing unnecessary attribute assignments for Boolean attributes, such as `read only` and `disabled`. In other words, it transforms `<input readonly="readonly">` to `<input readonly>`.

The `removeAttributeQuotes` option will remove quotes from attribute assignments where possible. This will transform `<div id="container"></div>` to `<div id=container></div>`.

Over the years, browsers have become better at heuristically determining the content between tags. As such, certain attributes may now be deemed redundant. By turning the `removeRedundantAttributes` option on, you can remove attributes such as `type="text/javascript"`.

Lastly, turn on `removeEmptyAttributes` to remove attributes assigned with empty strings.

13. The `grunt-usemin` plugin will be used to minimize and concatenate all JavaScript files involved in this project. The `usemin` plugin will involve the use of `grunt-contrib-concat`, `grunt-contrib-uglify`, and `grunt-rev` plugins. Define the configuration for the `usemin` plugin by adding the following code block:

```
config['useminPrepare'] = {
  options: {
    dest: 'dist'
  },
  html: 'src/index.html'
};

config['usemin'] = {
  options: {
    dirs: ['dist']
  },
  html: ['dist/{,*/}*.html']
};
```

Next, we move on by defining the `uglify` task, turning obfuscation off by adding the following code block after the declaration of `config`:

```
config['uglify'] = {
  options: {
    mangle: false
  }
};
```

Obfuscation is turned off via the `mangle` option as there are occasions when it may cause errors, thereby throwing exceptions and halting the JavaScript compilation process.

14. Now, modern browsers cache files locally using URLs to save the load time. This indicates that JavaScript, CSS, and even image files may be loading locally instead of from the official web server, leading to revision issues.

To solve this, it is advisable to cache bust your files by prepending a hashkey of the file to the filename. This way, any time a file is updated, its hash will subsequently be updated, causing an update to the filename. This is effectively **cache busting** the browser. To accomplish this with the JavaScript files in our project, we will add the following code block:

```
config['rev'] = {  
  files: {  
    src: [  
      'dist/scripts/{,*/}*.js',  
    ]  
  }  
};
```

This will prepend any scripts with a hashkey in our distributable project folder.

15. Lastly, you can complete the Gruntfile by replacing the task variable declaration with the following code:

```
var tasks = [  
  'clean',  
  'useminPrepare',  
  'htmlmin',  
  'concat',  
  'uglify',  
  'rev',  
  'usemin'  
];
```

These are the tasks that will run synchronously when the 'grunt build' command is run. It will start by cleaning the distributable folder as required. The useminPrepare command will initiate the start of the usemin process. Then, htmlmin, concat, uglify, and rev will be executed to minify the HTML and JavaScript files. The final usemin command will deliver an HTML file along with the minified, concatenated JavaScript file as required.

16. Now, you can scaffold the project by issuing the following command:

```
grunt build
```

This should produce a minified and concatenated version of your project in a dist (abbreviation for distributable) folder. Following this, you can open the website by running dist/index.html in your web browser.

Summary

To conclude, in this chapter, you have set up several of the binaries required for Grunt development. These will be thoroughly used in the projects introduced in the following chapters. In addition, the following chapters will introduce the advanced concepts surrounding Grunt.

The next chapter will cover a project that uses Jade, a templating engine, Sass, a CSS preprocessor, and the use of Grunt plugins to scaffold the project together.

2

Developing a Blog with Jade and Sass

This chapter will introduce you to the basic concepts of the Jade templating engine and will show you how to use it to implement a simple blog. This will help you begin to comprehend the underlying importance of minification and compression.

A brief summary of Jade

This chapter will discuss the background and underlying reasons behind the invention of Jade.

The following is a look into the official website for Jade:



Source: <http://jade-lang.com/>

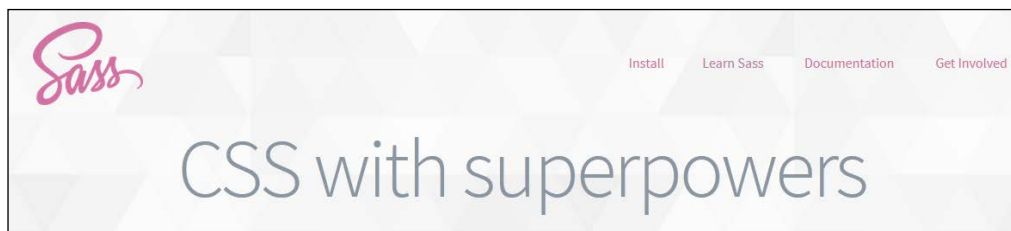
Jade is a templating engine built on Node.js (to be used for Node.js) and has since become fairly popular among the JavaScript community. It supports layouts, mixins, and embedded JavaScript. All of these features will be explored in this chapter. For instance, this chapter will cover the usage of mixins to generate comments and relevant posts for a given blog post.

The power behind Jade lies in its elegance. Using Jade, developers can now iterate with massive amounts of HTML templates without having to worry about overlapping code and lack of flexibility.

A brief summary of Sass

This chapter will briefly explain Sass as a CSS preprocessor, and how to leverage the framework to minify and concatenate style sheets.

The following is a look into the official website for Sass:



Source: <http://sass-lang.com/>

What is Sass?

Sass is a cascading style sheet (CSS) preprocessor; it is a tool that is used to scaffold, minify, and enhance the flexibility of the CSS framework. Like Jade, Sass supports mixins and modularization out of the box, features that will be taken advantage of in this chapter.

The primary reason behind the ubiquity of Sass among web developers today is its ability to split up a large CSS file into reasonably sized Sass files. Although loading many different CSS files is an option, load times of web pages are directly affected by the number of HTTP requests required to set them up. Scaffolding Sass files into CSS allows developers to leverage a platform that easily automates the process for them. For instance, splitting up the styles for an e-commerce website into `home.scss`, `item.scss` (for sold goods), and `contact.scss` could help developers delegate appropriate CSS classes for each type of web page.

Concatenation and minification

People will often ask what is so important about using tools such as Jade and Sass. To answer this, consider high-traffic websites such as blogs or e-commerce websites. With millions of hits a day, these websites must minimize the amount of HTTP requests required to load the page. This can be done through concatenating CSS and JavaScript files. Additionally, there exists a need to minimize the size of downloaded files to optimize the page load speed even further. Minification aids in this respect.

With web developers moving faster than ever, new changes are made to JavaScript and CSS repositories every day. Thus, the use of Grunt helps to optimize development efficiency by automating away large portions of the DevOps integration process.

Building the blog

We will now start the second project that will involve the basics of Jade and Sass. It will also incorporate many Grunt plugins, such as `grunt-contrib-watch`, `grunt-contrib-jade`, and `grunt-contrib-compass`, which will be introduced and explained throughout the chapter.

Installing the required Grunt plugins

You can install the Grunt.js plugins by performing the following steps:

1. Firstly, visit <https://github.com/packt-mg/Chapter-2> for the project template.
2. Following this, in a terminal, traverse to a directory of your choice that will be used to store the project and its contents.
3. In your web browser, copy the HTTPS clone URL of the project, as shown in the following screenshot:



4. Copy the HTTPS clone URL from the input box.

5. Issue a Git clone command by pasting the HTTPS clone URL in the terminal, as shown in the following command:
git clone <HTTPS clone url>
6. Traverse into the recently cloned directory by issuing the following command:
cd Chapter-2
7. Install the Node dependencies as shown in the following command:
npm install



Note that several newly introduced, important Grunt plugins will have to be installed after this step. The `grunt-contrib-watch` plugin will be used to automate the compilation of both Jade and Sass files by linking to the `grunt-contrib-jade` and `grunt-contrib-compass` plugins respectively. The automated compilation will occur when either set of files are saved.

8. You should then install the Bower dependencies by issuing the following command:

bower install

This will install Bootstrap, a lightweight but powerful user interface framework. Specifically, Bower will install it in the form of Sass files.

Configuring grunt-contrib-watch

In order to configure `grunt-contrib-watch`, perform the following steps:

1. The first task is to set up `grunt-contrib-watch`. Copy the following code into your Gruntfile after the initialization of the `config` object:

```
var config = {};  
  
config['watch'] = {  
  options: {  
    nospawn: true  
  },  
  compass: {  
    files: ['src/styles/{,*/}*.scss,sass'],  
    tasks: ['compass:server']  
  },  
  jade: {  
    files: ['src/templates/{,*/}*.jade'],  
    tasks: ['jade:server']  
  }  
};
```

Note that the `nospawn` option helps speed up the scaffolding process by around 500 milliseconds on most machines. This is possible by restricting the spawning of child processes for each task that is run.

The `grunt-contrib-watch` plugin expects a series of individual tasks, each with a set of watched files. In our case, this configuration is watching for Sass files running the `grunt-contrib-compass` task whenever a Sass file is created or changed. In addition, the Gruntfile also watches for Jade files running the `grunt-contrib-jade` task whenever a Jade file is created or changed.



The `{,*/}* .jade` expression matches Jade files in the root directory and all of its subsequent subdirectories. Likewise, the `{,*/}*.{scss,sass}` expression matches for all SCSS/Sass files in the same manner. This can be understood when you interpret the `*` character as a wildcard for any string and know that the expression will match any element found within the `{ }` braces.

2. Next, you will need to define the `grunt-contrib-compass` task for the `grunt-contrib-watch` task to call. Add the following block of code anywhere after the initialization of the `config` object:

```
config['compass'] = {
  options: {
    sassDir: 'src/styles/sass',
    cssDir: 'src/styles',
    importPath: 'src/bower_components',
    relativeAssets: false
  },
  dist: {},
  server: {}
};
```



It is important to note that the `sassDir` variable stores the directory where all user-built Sass files can be found. It is also where the main Sass file will be found for this particular project.

The `cssDir` variable stores the directory where the compiled CSS file will be located after the `grunt-contrib-compass` task runs and scaffolds all the Sass files together.

The `importPath` variable stores a separate directory for which Sass can search when an imported file is required for compilation. In this case, you will include all files within the Bower-generated `bower_components` folder. This folder will contain the recently installed `bootstrap-sass` package, an import for this project.

Setting the last option `relativeAssets` to `false` will override the generation of relative paths for assets such as images or fonts. This will minimize the 404 errors for the sake of this project.

Lastly, you should be aware that the `dist` and `server` objects represent `compass:dist` and `compass:server` respectively and are separate task configurations of the `grunt-contrib-compass` plugin itself. In this case, `compass:dist` will be used for the deployment of the project into a distributable format. On the other hand, `compass:server` will be used for on-the-fly compilation through the `grunt-contrib-watch` plugin. Note that since both are empty objects, there is virtually no difference in configuration between the two tasks. However, it is still the best practice to segment your tasks out like this in case a specific configuration setting is required by one task or another.

3. It is important to note that the `grunt-contrib-watch` plugin configuration also requires a configuration for the `grunt-contrib-jade` plugin. Add the following code block anywhere after the initialization of the `config` object:

```
config['jade'] = {
  dist: {
    files: {
      'src/index.html':
'src/templates/index.jade',
      'src/golden-dragon.html':
'src/templates/golden-dragon.jade',
      'src/little-pizzeria.html':
'src/templates/little-pizzeria.jade'
    }
  },
  server: {
    options: {
      data: {
        debug: false
      }
    },
    files: {
      'src/index.html':
'src/templates/index.jade',
      'src/golden-dragon.html':
'src/templates/golden-dragon.jade',
      'src/little-pizzeria.html':
'src/templates/little-pizzeria.jade'
    }
  }
};
```

At first glance, you will note that this configuration is not similar to the others. The `grunt-contrib-jade` plugin maps HTML files to their respective Jade templates via a `files` object. In this case, this configuration will automate the compilation of `index.html`, `golden-dragon.html`, and `little-pizzeria.html`. The Jade files, however, will have several other template dependencies that will not be explicitly referenced in this Gruntfile. These dependencies will be covered later in this chapter.

4. Finally, round this off by going into your terminal and running the following command:

```
grunt watch
```

This will start the `grunt-contrib-watch` daemon, running the previously defined tasks whenever Jade or Sass files change. Leave this running in the background.

Developing the blog

Using the previously installed Sass and Jade frameworks, you can now begin to develop your blog by carrying out the following steps:

1. Traverse to your Sass file directory located at `src/styles/sass`.
2. Within this directory, you will find `main.scss` and `_bootstrap_overrides.scss`. Populate your `main.scss` file with the following imports:

```
@import 'bootstrap-sass/lib/bootstrap.scss';
@import '_bootstrap-overrides.scss';
```

This will import Bootstrap, the aforementioned CSS framework, and a set of overrides that will customize the blog to your liking. You should note that when you save `main.scss`, the terminal running the `grunt watch` should populate with an update regarding the compilation of `main.css`, which can be found in `src/styles`.

3. Since Bootstrap is only a starting point, it is important to customize our blog on top of it. Add the following lines of code into `_bootstrap-overrides.scss`:

```
body {
  margin-top: 100px;
}
```

```
footer {
  margin: 50px 0;
}
```



The underscore found in the filename `_bootstrap-overrides.scss` ensures that the file is labeled as an import. This ensures that the `grunt-contrib-compass` task and all subsequent Sass compilers will ignore `_bootstrap-overrides.scss` during the compilation process and will only use it as an import for officially compiled Sass files.

4. Next, you will look into implementing the project's Jade templates. Traverse to your Jade template directory located at `src/templates`.
5. It is important to note that there are two Jade layouts built already. A Jade layout is a template used across many different pages. In this case, `post.jade` is the layout template used for all blog posts for this project, whereas `blog.jade` is used as the layout template for the main blog page itself. Looking at the main `blog.jade` layout, you should see the following:

```
doctype html
//if lt IE 7
  html.no-js.lt-ie9.lt-ie8.lt-ie7
//if IE 7
  html.no-js.lt-ie9.lt-ie8
//if IE 8
  html.no-js.lt-ie9
// [if gt IE 8] <![endif]
html.no-js
  // <![endif]
  head
    meta(charset='utf-8')
    meta(http-equiv='X-UA-Compatible', content='IE=edge,chrome=1')
    title Chapter 2 - Tasty Eats Blog
    meta(name='description', content='')
    meta(name='viewport', content='width=device-width')
    link(rel='stylesheet', href='styles/main.css')
  body
    .container
      h1
        a(href='index.html') Tasty Eats
      .row
        .col-lg-12
          block posts
      hr
      footer
        .row
          .col-lg-12
p Copyright © Tasty Eats
```

For reference, the preceding segment of code is converted to HTML, which is as follows:

```
<!DOCTYPE html><!-- [if lt IE 7]>
<html class="no-js lt-ie9 lt-ie8 lt-ie7"></html><![endif]--><!--
[if IE 7]>
<html class="no-js lt-ie9 lt-ie8"></html><![endif]--><!-- [if IE
8]>
<html class="no-js lt-ie9"></html><![endif]-->
<!-- [if gt IE 8] <!-->
<html class="no-js">
  <!-- <![endif]-->
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">
    <title>Chapter 2 - Tasty Eats Blog</title>
    <meta name="description" content="">
    <meta name="viewport" content="width=device-width">
    <link rel="stylesheet" href="styles/main.css">
  </head>
  <body>
    <div class="container">
      <h1><a href="index.html">Tasty Eats</a></h1>
      <div class="row">
        <div class="col-lg-12">
          </div>
        </div>
        <hr>
        <footer>
          <div class="row">
            <div class="col-lg-12">
              <p>Copyright © Tasty Eats</p>
            </div>
          </div>
        </footer>
      </div>
    </body>
  </html>
```

You should note that Jade removes the need for tag completion as the engine will infer where to put tags based on the amount of indents before the given tag in the file. Any references after a period such as `.col-lg-12` represent a divider with the reference as its class. For example:

```
<div class="col-lg-12"></div>
```

Likewise, other attributes can be defined within parentheses after the tag declaration itself. For instance, `meta(name='viewport', content='width=device-width')` becomes `<meta name="viewport" content="width=device-width">`.

Lastly, it is important to note the block posts tag in the template. When a layout is extended, the root Jade template can populate any blocks defined in the layout with anything it wants. This will be covered in the next step.

6. Before jumping into the implementation of the blog, it is important to define mixins and what they do in a Jade context. In our case, mixins will be used in their simplest form by individually defining them in separate files. Open `src/templates/mixins/posts.jade` and add the following code block into the file, keeping track of the indentation:

```
mixin posts(collection)
  each post in collection
    h2
      a(href=post.link) #{post.title}
    p.lead
      | by #{post.author}
    hr
    p
      span.glyphicon.glyphicon-time
      | #{post.date}
    hr
    a(href=post.link)
      img.img-responsive(src=post.image)
    hr
    br
```

This defines the `posts` mixin that will be used to implement the blog template. This mixin takes an array of objects and traverses through it in a `foreach` loop, printing out a partial representing a given blog post.

7. Next, the blog template will be populated with the use of the newly defined posts mixin. Add the following code block to your `index.jade` file:

```
extends layouts/blog.jade

include mixins/posts.jade

block posts
  - collection = []
  - collection.push({title: 'Golden Dragon Review', author:
    'Wolfgang Amadeus Mozart', date: 'Posted on January 21st,
    2014 at 9:00 PM', image: 'images/golden-dragon.png', link:
    'golden-dragon.html'})
  - collection.push({title: 'Little Pizzeria Review',
    author: 'Ludwig Van Beethoven', date: 'Posted on January
    18th, 2014 at 6:00 PM', image: 'images/little-
    pizzeria.png', link: 'little-pizzeria.html'})
  mixin posts(collection)
```



Note that the lines of code prepended with a dash symbol (-) must be on one line. In other words, the following code block should be typed out on the same line:

```
- collection.push({title: 'Little Pizzeria Review',
author: 'Ludwig Van Beethoven', date: 'Posted on
January 18th, 2014 at 6:00 PM', image: 'images/
little-pizzeria.png', link: 'little-pizzeria.html'})
```

These lines embed JavaScript to the template and are used to define the data that is to be presented on the blog. Once you save this file, your grunt watch task should trigger and compile your `index.jade` file into an `index.html` file.

8. It is now time to move on toward implementing the individual post pages. To do this, it is important to define the necessary mixins for our `post.jade` layout. Open your `popular.jade` file, located at `src/templates/mixins/popular.jade`.

Add the following code block to the file:

```
mixin popular(collection)
  .row
    hr
    .col-lg-6
      ul.list-unstyled
        each post in collection
          li
            a(href=post.link) #{post.text}
```

This mixin will be in charge of showing other popular blog posts on a given blog post page. Again, it utilizes a `foreach` loop to accomplish the task, iterating through an array of objects.

9. Next, open up your `comments.jade` file located at `src/templates/mixins/comments.jade` and add the following code block to the file:

```
mixin comments(collection)
  each comment in collection
    h3
      | #{comment.author}
    br
    small #{comment.date}
  p
    | #{comment.text}
```

This particular template file is in charge of printing reader-submitted comments at the end of a blog post. It also utilizes a `foreach` loop again, taking in a collection (an array of objects) as a parameter.

10. Now that the required mixins are defined, it is possible to extend the `post.jade` template as required. Open up `golden-dragon.jade` located at `src/templates/golden-dragon.jade` and then add the following code block to the file:

```
extends layouts/post.jade

include mixins/comments.jade
include mixins/popular.jade

block title
  | Golden Dragon Review

block author
  | Wolfgang Amadeus Mozart

block date
  | Posted on January 21st, 2014 at 9:00 PM

block image
  img.img-responsive(src='images/golden-dragon.png')

block content
  | You won't be disappointed.
block popular
  - collection = []
  - collection.push({ link: 'little-pizzeria.html', text:
'Little Pizzeria' });
  mixin popular(collection)

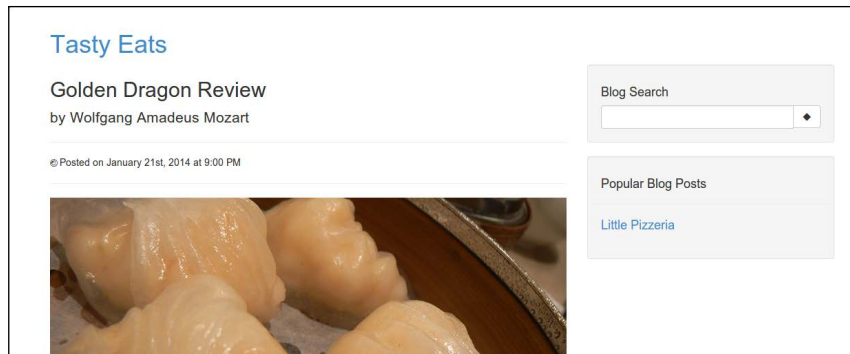
block comments
  - collection = [];
  - collection.push({ author: 'Ludwig van Beethoven', date:
'10:38 PM on January 21st, 2014', text: 'Thanks for the
information!' });
  mixin comments(collection)
```



Note that the lines of code prepended with the dash symbol (-) must be on one line. In other words, the following code block should be typed out on the same line:

```
- collection.push({ link: 'little-pizzeria.html',
text: 'Little Pizzeria' });
```

Once you save this file, the grunt watch task should trigger once again, compiling the Jade file into an HTML file. Open the compiled `golden-dragon.html` file in your favorite web browser, and you should see something similar to the following screenshot:



11. Lastly, it is important to populate the final blog post. Open up `little-pizzeria.jade` located at `src/templates/little-pizzeria.jade`. Then, add the following code block to the file:

```
extends layouts/post.jade

include mixins/comments.jade
include mixins/popular.jade

block title
  | Little Pizzeria Review

block author
  | Ludwig Van Beethoven

block date
  | Posted on January 18th, 2014 at 6:00 PM

block image
  img.img-responsive(src='images/little-pizzeria.png')

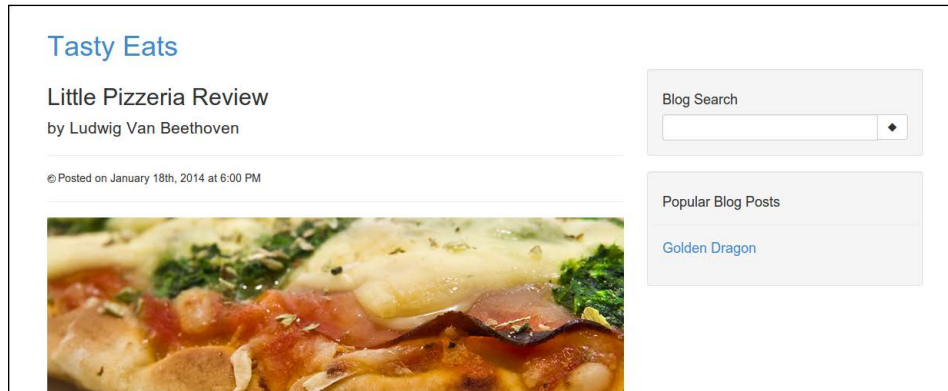
block content
  | Very fast delivery.

block popular
  - collection = []
  - collection.push({ link: 'golden-dragon.html', text:
'Golden Dragon' });
  mixin popular(collection)

block comments
  - collection = [];
  mixin comments(collection)
```

As with all the previous templates, ensure that all embedded JavaScript (the lines prepended with dash symbols) are on the same line.

Once you save this file, open `little-pizzeria.html` in your web browser, and you should see something similar to this:



12. Congratulations! You have just finished the development portion of this project. It is now time to scaffold this project into a distributable package. Open up your `Gruntfile.js` file in the root directory.

Implementing the custom build Grunt task

To deploy your blog to a production server, one must define a custom build task to automate the DevOps integration as follows:

1. Add the following code block anywhere after the initialization of the `config` variable:

```
config['clean'] = {
  build: {
    files: [{
      dot: true,
      src: [
        'dist/*',
        '!dist/.git*'
      ]
    }]
  }
};
```

Just like the configuration used in *Chapter 1, Getting Started*, this configures the `grunt-contrib-clean` plugin to clean out the distributable folder when required.

2. Add the following code block to your Gruntfile:

```
config['htmlmin'] = {
  dist: {
    options: {
      collapseBooleanAttributes: true,
      removeAttributeQuotes: true,
      removeRedundantAttributes: true,
      removeEmptyAttributes: true
    },
    files: [{
      expand: true,
      cwd: 'src',
      src: '{,*/}*.html',
      dest: 'dist'
    }]
  }
};
```

Again, like with the first chapter, this code block will help you to configure the `grunt-contrib-htmlmin` plugin as required. The `src` attribute, set as `{,*/}*.html`, helps to match for all HTML files in all subdirectories of the `src` folder.

3. It is now time to incorporate the configuration for the `grunt-contrib-copy` plugin. You can do this by adding the following code block to your Gruntfile:

```
config['copy'] = {
  dist: {
    files: [{
      expand: true,
      dot: true,
      cwd: 'src',
      dest: 'dist',
      src: [
        'images/{,*/}*.png'
      ]
    }]
  }
};
```

The primary use of this plugin will be to move all images from the `src` directory to the `dist` directory when building. The `expand` option will create a directory, if required, when the directory's parent does not exist. Thus, by setting `expand` to `true`, `dist/images` can be created as a directory even if `dist` does not initially exist as one.

4. Lastly, it is important to consider the minification of our compiled CSS files. To do this, you can configure your `grunt-contrib-cssmin` task with the following code block:

```
config['cssmin'] = {
  dist: {
    files: {
      'dist/styles/main.css': [
        'src/styles/{,*/}*.css'
      ]
    }
  }
};
```

Note that this configuration will set the default behavior of the `grunt-contrib-cssmin` task to concatenate and minify all CSS files within the `styles` directory in the `src` folder and push the results to the `main.css` file in the `dist` directory inside the `styles` folder.

5. Finally, it is time to define the build task. Replace the initialization of the tasks array `var tasks = []`; with the following code:

```
var tasks = [
  'clean',
  'jade:dist',
  'compass:dist',
  'htmlmin:dist',
  'copy',
  'cssmin'
];
```

This task order will clean the `dist` directory, compile all Jade and Sass files, and minify the generated HTML files. Then, it will copy all images and HTML/CSS files from the `src` directory to the `dist` directory.

6. Now, you can open up a terminal and finally scaffold your project by issuing the following command:

grunt build

Following this, you can open the website by running `dist/index.html` in your web browser.

Summary

This chapter covered the use of the `grunt-contrib-watch` daemon to automate compilation during development. It also covered the importance of automated compression, minification, and compilation in the web development world today. This concept will be referenced in the next chapter, automating the compilation of CoffeeScript alongside Jade and Sass.

3

Making an Employee Management System

This chapter will investigate the use of CoffeeScript and its unique class system to implement an employee management system that you can use to fire, hire, and promote employees.

A brief summary of CoffeeScript

CoffeeScript provides an alternative to programming in JavaScript, enforcing its best practices by stripping away certain language features and forcing design constructs. This chapter will explore these concepts.

The following is a look into the official website for CoffeeScript:



Source: <http://coffeescript.org/>

Building the employee management system

We will now start the third project which will demonstrate the flexibility of CoffeeScript and how Grunt can streamline JavaScript development.

Installing the required Grunt plugins

You can install the Grunt.js plugins by performing the following steps:

1. First, visit <https://github.com/packt-mg/Chapter-3> to access the project template.
2. Following this, in a terminal, traverse to a directory of your choice which will be used to store the project and its contents.
3. In your web browser, copy the HTTPS clone URL of the project, as shown in the following screenshot:



You should copy this HTTPS clone URL from the input box.

4. Following this, issue a Git clone command by pasting the HTTPS clone URL in the terminal, as shown in the following command:
5. Then, you can traverse into the recently cloned directory by issuing the following command:

```
git clone <HTTPS clone url>
```

```
cd Chapter-3
```

You can then install the Node dependencies, as shown:

```
npm install
```



This iteration, `grunt-contrib-coffee`, will be used to automate the compilation of both the CoffeeScript files. The `grunt-contrib-watch` plugin will be used alongside this plugin and `grunt-contrib-sass` to automate the production of CSS and JavaScript files.

6. Finally, you should then install the Bower dependencies by issuing the following command:

```
bower install
```

This will install Bootstrap, a library that was used in *Chapter 2, Developing a Blog with Jade and Sass*. Additionally, jQuery will be installed for the use of this project.

Configuring grunt-contrib-watch

In order to configure `grunt-contrib-watch`, you should carry out the following steps:

1. The first task is to set up `grunt-contrib-watch`. Copy the following code into your Gruntfile after the initialization of the `config` object:

```
var config = {};

config['watch'] = {
  options: {
    nospawn: true
  },
  coffee: {
    files: ['src/coffee/{,*/}*.coffee'],
    tasks: ['coffee:server']
  },
  compass: {
    files: ['src/styles/{,*/}*.scss,sass'],
    tasks: ['compass:server']
  }
};
```

In this case, this configuration is watching for Sass and Coffee files, running the `grunt-contrib-compass` and `grunt-contrib-coffee` tasks whenever a Sass or Coffee file is created or changed, respectively.

2. Next, the `grunt-contrib-compass` task from *Chapter 2, Developing a Blog with Jade and Sass*, will be reused. Add the following block of code anywhere after the initialization of the `config` object:

```
config['compass'] = {
  options: {
    sassDir: 'src/styles/sass',
    cssDir: 'src/styles',
    importPath: 'src/bower_components',
    relativeAssets: false
  },
  dist: {},
  server: {}
};
```

3. The `grunt-contrib-coffee` task will now be defined. Add the following block of code anywhere after the initialization of the `config` object:

```
config['coffee'] = {
  dist: {
    files: [{
      expand: true,
      cwd: 'src/coffee',
      src: '{,*/}*.coffee',
      dest: 'dist/scripts',
      ext: '.js'
    }]
  },
  server: {
    files: [{
      expand: true,
      cwd: 'src/coffee',
      src: '{,*/}*.coffee',
      dest: 'src/scripts',
      ext: '.js'
    }]
  }
};
```

It should be noted that CoffeeScript files will be located in `src/coffee` and compiled into `src/scripts` by `grunt-contrib-watch`.

4. Finally, go into your terminal and run the following command:

```
grunt watch
```

This will start the `grunt-contrib-watch` daemon, running the previously defined tasks whenever CoffeeScript or Sass files change. Leave this running in the background.

Developing the employee management system

This section will investigate the development of a simple employee management system with CoffeeScript. Perform the following steps:

1. The first thing you should do is traverse to your Sass file directory located at `src/styles/sass`.
2. Within this directory, you will find `main.scss`. Populate your `main.scss` file with the following import:

```
@import 'bootstrap-sass/lib/bootstrap.scss';
```

As shown in *Chapter 2, Developing a Blog with Jade and Sass*, this will import Bootstrap. You should note that when you save `main.scss`, the terminal running the Grunt watch task should populate with an update regarding the compilation of `main.css`, which can be found in `src/styles`.

3. The CoffeeScript files will now be populated. Traverse to `src/coffee/app` and open `init.coffee`. You can then populate the file by inputting the following line:

```
window.GLOBALS = {}
```



Although this is technically CoffeeScript, the line of code is identical to its compiled output; the `GLOBALS` object will be used to import and export modules in CoffeeScript. This is because CoffeeScript enforces JavaScript's best practices, wrapping every script in its own functional scope. Thus, variables from other files are hidden.

4. Following this, open `main.coffee` and populate the file by adding the following lines of code:

```
Employee = window.GLOBALS.Employee
Panel = window.GLOBALS.Panel

$ ->
  collection = [
    {
      name: "John"
      level: 4
      hours: 40
    }
    {
      name: "Jane"
      level: 4
      hours: 40
    }
    {
      name: "Max"
      level: 4
      hours: 40
    }
  ]

  employees = {}

  for model in collection
    employees[model.name] = new Employee model.name,
model.level, model.hours

  panel = new Panel $("#table-employee"), employees
```



Note that CoffeeScript supports various syntactic sugar including indentation rules, foreach loops, and the removal of required commas and parentheses.

This code block actually compiles to the following equivalent JavaScript file:

```
var Employee, Panel;

Employee = window.GLOBALS.Employee;

Panel = window.GLOBALS.Panel;

$(function() {
    var collection, employees, model, panel, _i, _len;
    collection = [
        {
            name: "John",
            level: 4,
            hours: 40
        }, {
            name: "Jane",
            level: 4,
            hours: 40
        }, {
            name: "Max",
            level: 4,
            hours: 40
        }
    ];
    employees = {};
    for (_i = 0, _len = collection.length; _i < _len; _i++) {
        model = collection[_i];
        employees[model.name] = new Employee(model.name, model.level,
model.hours);
    }
    return panel = new Panel($("#table-employee"), employees);
});
```

Lastly, it is important to note that both the `Employee` and `Panel` classes yet to be defined. These will be defined in the following step.

5. Navigate to `src/coffee/models` and open `person.coffee`.

The `Person` class will be the superclass for the `Employee` class that is used throughout the project. Add the following lines of code into the file:

```
class Person
  constructor: (@name) ->
    throw "Invalid Name" if @name.length is 0

  getName: ->
    @name

window.GLOBALS.Person = Person
```

This helps illustrate one of CoffeeScript's most important value propositions – its class system. This also helps to alleviate the confusion surrounding JavaScript's use of prototypes to represent classes for many novices. In compiled JavaScript, the previous code block would look like the following:

```
var Person;

Person = (function() {
  function Person(name) {
    this.name = name;
    if (this.name.length === 0) {
      throw "Invalid Name";
    }
  }

  Person.prototype.getName = function() {
    return this.name;
  };

  return Person;
})();

window.GLOBALS.Person = Person;
```

6. You should then open `employee.coffee`, and take note of the fact that the `Employee` class will represent the internal implementation of each employee operation. Add the following lines of code into the file:

```
Person = window.GLOBALS.Person

class Employee extends Person
  constructor: (@name, @level, @hours) ->
    super(@name)
    throw "Invalid Employee Level" if @level < 1
    throw "Invalid Allocated Hours" if @hours < 0

  getLevel: ->
    @level

  getHours: ->
    @hours

  isFired: ->
    @level is 0

  promote: ->
    @level++
    @hours += 4

  fire: ->
    @level = 0
    @hours = 0

window.GLOBALS.Employee = Employee
```

Note that CoffeeScript implements the `super` function found in Java and other object-oriented languages.

7. Following this, open `panel.coffee`. Again, take note that the `Panel` class will be used to represent all events related to the employee management panel.

Add the following lines of code into the file:

```
Employee = window.GLOBALS.Employee

class Panel
  constructor: (@panel, @employees) ->
    @init()

  init: ->
    _this = this

    @render()

    $("#btn-hire").click ->
      key = $("#input-employee").val()
      _this.employees[key] = new Employee key, 1, 40
      _this.render()

  getEmployees: ->
    @employees

  setEmployees: (employees) ->
    @employees = employees

  render: ->
    _this = this

    @panel.find("tbody").html ""

    for key, value of @employees
      if value.isFired()
        continue

      row =
        "<tr data-id='#{key}'>
          <td>#{key}</td>
          <td>#{value.level}</td>
          <td>#{value.hours}</td>
          <td>
            <button class='btn btn-primary btn-promote'
data-id='#{key}'>Promote</button>
            <button class='btn btn-danger btn-fire' data-
id='#{key}'>Fire</button>
          </td>
        </tr>"
      @panel.find("tbody:last").append(row)

    @panel.find(".btn-promote").each ->
      $(this).click ->
```

```

        key = $(this).attr("data-id")
        _this.employees[key].promote()
        _this.render()

        @panel.find(".btn-fire").each ->
        $(this).click ->
        key = $(this).attr("data-id")
        _this.employees[key].fire()
        _this.render()

window.GLOBALS.Panel = Panel

```



Ensure that proper indentation is upheld throughout the file. CoffeeScript's flexibility comes from its strict indentation rules, so the compilation process will likely break if they are not upheld.

You should note that while filling all of these CoffeeScript files, the `grunt-contrib-watch` daemon has been running in the background, compiling each one on save. Now that all of the files have been populated, open `index.html` in a web browser and you should see something similar to the following:

Employee Management System			
Name	Position Level	Hours	Actions
John	4	40	Promote Fire
Jane	4	40	Promote Fire
Max	4	40	Promote Fire
<input type="text"/>		Hire	

As a final note in the `index.html` file, observe the following lines:

```

<!-- build:js scripts/compiled.js -->
<script type="text/javascript" src="bower_components/jquery/
jquery.js"></script>
<script type="text/javascript" src="scripts/app/init.js"></
script>
<script type="text/javascript" src="scripts/models/person.
js"></script>
<script type="text/javascript" src="scripts/models/employee.
js"></script>
<script type="text/javascript" src="scripts/models/panel.
js"></script>
<script type="text/javascript" src="scripts/app/main.js"></
script>
<!-- endbuild -->

```

These lines of code will be particularly important for the custom build Grunt task that will be implemented in the next section. The `grunt-usemin` plugin will take this list of scripts and build it into a final `compiled.js` JavaScript file.

Implementing the custom build Grunt task

The following steps will demonstrate how to set up a build task for this particular project in Grunt:

1. Open the `Gruntfile.js` file. Initially, the `grunt-contrib-clean` task will be reused. You should then add the following lines of code after the initialization of the `config` object:

```
config['clean'] = {
  build: {
    files: [{
      dot: true,
      src: [
        'dist/*',
        '!dist/.git*'
      ]
    }]
  }
};
```

2. As the `grunt-usemin` task will be used for this configuration, it will be defined in the following block. The `grunt-usemin` task helps to minify and concatenate a list of JavaScript or CSS files. As our CSS is already being minified and concatenated via the Sass preprocessor, our configuration will only use the tasks required for JavaScript optimization. A preparation task, `useminPrepare`, will be first defined to specify the source HTML file and will be followed up with the `usemin` task to define the final destination of the compiled HTML file. Add the following block of code to the `Gruntfile`:

```
config['useminPrepare'] = {
  options: {
    dest: 'dist'
  },
  html: 'src/index.html'
};

config['usemin'] = {
  options: {
    dirs: ['dist']
  },
  html: ['dist/{,*/}*.html']
};
```

3. Next, the `grunt-htmlmin-contrib` task will be set up. You should add the following code block to your Gruntfile:

```
config['htmlmin'] = {
  dist: {
    options: {
      collapseBooleanAttributes: true,
      removeAttributeQuotes: true,
      removeRedundantAttributes: true,
      removeEmptyAttributes: true
    },
    files: [{
      expand: true,
      cwd: 'src',
      src: '{,*/}*.html',
      dest: 'dist'
    }]
  }
};
```

Again, similar to before, this code block will minify the amount of HTML that is required.

4. As required by `grunt-usemin`, the `grunt-contrib-uglify` task will be defined in this step. You should add the following block to your Gruntfile:

```
config['uglify'] = {
  options: {
    mangle: false
  }
};
```

5. The `grunt-contrib-copy` task will need to be defined for the last stage of the build task. You can achieve this by adding the following block to your Gruntfile:

```
config['copy'] = {
  build: {
    files: [{
      expand: true,
      dot: true,
      cwd: 'src',
      dest: 'dist',
      src: []
    }]
  }
};
```

6. In order to avoid conflict with the browser cache, the compiled files used for this project will need to have a cache bust prepended to their filenames. This can be done by adding the following block of code to your Gruntfile:

```
config['rev'] = {
  dist: {
    files: {
      src: [
        'dist/scripts/{,*/}*.js',
      ]
    }
  }
};
```

7. Finally, it is important to configure your `grunt-contrib-cssmin` task by inserting the following section of code:

```
config['cssmin'] = {
  dist: {
    files: {
      'dist/styles/main.css': [
        'src/styles/{,*/}*.css'
      ]
    }
  }
};
```

This is concluded by defining the build task. Replace the initialization of the tasks array that contains `var tasks = []` with this whole section of code:

```
var tasks = [
  'clean:build',
  'useminPrepare',
  'htmlmin',
  'cssmin',
  'concat',
  'uglify',
  'copy',
  'rev',
  'usemin'
];
```

What will happen now is that this task order will clean the `dist` directory, compile all CoffeeScript and Sass files, and minify the generated HTML files.

8. Now, you can open up a terminal and finally scaffold your project by issuing the following command:

```
grunt build
```

Following this, you can open the website by running `dist/index.html` in your web browser.

Summary

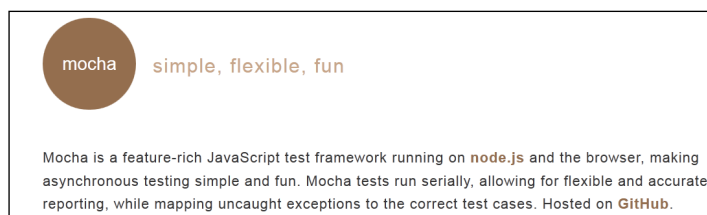
This chapter covered the use of the `grunt-contrib-coffee` daemon to automate the compilation of CoffeeScript files. By covering the syntactic sugar provided by CoffeeScript, one can easily see its advantages over JavaScript. This concept, including the concepts from all of the previous chapters, will be used heavily in the upcoming chapter as we further implement scripts and tests in CoffeeScript.

4

Final Project – Simple Bulletin Board System

This chapter will combine the concepts used in the previous three projects into a final project. It will also introduce the Mocha testing library and look at how to use it alongside Grunt to automate testing prior to distribution, throughout the build process.

The official logo of Mocha is as follows:



Source: <http://visionmedia.github.io/mocha/>

Mocha is a testing framework known by many for its feature set, flexibility, and ability to render on both the browser and command line. Its feature set supports testing of asynchronous code, coverage tooling and benchmarking. Using CoffeeScript for our tests, this chapter will utilize Zombie as a headless web client to test the project. Lastly, the Node.js standard assertion library will be used to ensure that the expected functionality occurs throughout our website.



A headless web client can browse the Internet like any other web browser. It includes basic functionalities such as the ability to fetch HTML, issue HTTP requests, and run JavaScript (given a headless script such as PhantomJS). However, it does not have rendering functionality. This means that certain testing functionalities, such as determining if a DOM element is visible or not, are unavailable.

Installing the required Grunt plugins

You can install the Grunt.js plugins by performing the following steps:

1. Firstly, access the project template at <https://github.com/packt-mg/Chapter-4> to access the project template.
2. Following this, in a terminal, traverse to a directory of your choice, which will be used to store the project and its contents.
3. In your web browser, copy the HTTPS clone URL of the project, as shown in the following screenshot:



You should copy this HTTPS clone URL from the input box.

4. Following this, issue a Git clone command by pasting the HTTPS clone URL in the terminal, as shown in the following command:

```
git clone <HTTPS clone url>
```

5. Then, traverse into the recently cloned directory by issuing the following command:

```
cd Chapter-4
```

You can now install the Node dependencies, as shown in the following command:

```
npm install
```



Mocha and Zombie are all npm packages and will be installed via this project's `package.json` file. The Node.js assertion library comes with the npm installation covered in *Chapter 1, Getting Started*.

This iteration `grunt-contrib-connect` will be used to push our local web files to a Grunt server. This will allow our Mocha tests to run on a local server. The `grunt-mocha-test` task and Zombie will be installed to complete the testing suite. Lastly, many of the plugins found in the previous chapters will be reintroduced for this final project.

6. Finally, install the Bower dependencies, by issuing the following command:

```
bower install
```

This will install Bootstrap and jQuery, libraries used throughout the previous chapters.

Configuring grunt-contrib-watch

In order to configure `grunt-contrib-watch`, carry out the following steps:

1. Copy the following code into your Gruntfile after the initialization of the config object:

```
var config = {};

config['watch'] = {
  options: {
    nospawn: true
  },
  coffee: {
    files: ['src/coffee/{,*/}*.coffee'],
    tasks: ['coffee:server']
  },
  compass: {
    files: ['src/styles/{,*/}*.scss,sass'],
    tasks: ['compass:server']
  },
  jade: {
    files: ['src/templates/{,*/}*.jade'],
    tasks: ['jade:server']
  }
};
```

In this case, the configuration is watching for changes to the repository's CoffeeScript, Sass, and Jade files.

2. Next, the `grunt-contrib-compass`, `grunt-contrib-coffee`, and `grunt-contrib-jade` tasks from the previous chapters will be reused. Add the following block of code anywhere after the initialization of the `config` object:

```
config['compass'] = {
  options: {
    sassDir: 'src/styles/sass',
    cssDir: 'src/styles',
    importPath: 'src/bower_components',
    relativeAssets: false
  },
  dist: {},
  server: {}
};

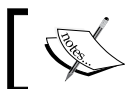
config['jade'] = {
  dist: {
    options: {
      pretty: true
    },
    files: {
      'src/index.html': 'src/templates/index.jade'
    }
  },
  server: {
    options: {
      data: {
        debug: false
      },
      pretty: true
    },
    files: {
      'src/index.html': 'src/templates/index.jade'
    }
  }
};

config['coffee'] = {
  dist: {
    files: [{
      expand: true,
      cwd: 'src/coffee',
      src: '{,*/}*.coffee',
      dest: 'dist/scripts',
      ext: '.js'
    }]
  },
  server: {
    files: [{
      expand: true,
```

```

        cwd: 'src/coffee',
        src: '{,*/}*.coffee',
        dest: 'src/scripts',
        ext: '.js'
      }
    }
  };

```



As before, the `dist` and `server` tasks are defined for each Grunt plugin in order to set up the watch and build Grunt tasks, respectively.

3. Finally, go into your terminal and run the following command:

```
grunt watch
```

This will start the `grunt-contrib-watch` daemon, running the previously defined tasks whenever CoffeeScript, Sass, or Jade files change. Leave this running in the background. Going forward, it is advisable to open another terminal window for this project.

Developing a simple Bulletin Board System (BBS)

This section will investigate the development of a simple employee management system. Perform the following steps:

1. The first thing you should do is move to your Sass file directory, located in `src/styles/sass`.
2. Within this directory, you will find `main.scss`. Populate your `main.scss` file with the following imports:

```

@import 'bootstrap-sass/lib/bootstrap.scss';
@import '_bbs.scss';

```

Note that when you save `main.scss`, the terminal running the Grunt watch task populates with an update regarding the compilation of `main.css`, which can be found in `src/styles`.

3. Following this, open `_bbs.scss` and add the following lines of code:

```
#board {  
  .post {  
    border-radius: 0;  
    padding-bottom: 32px;  
  }  
  
  .reply {  
    padding: 0 19px 0 40px;  
  }  
}
```

This will add the padding changes required to create an appealing bulletin board. Note that Sass's use of embedded CSS attributes allows the previous code block to be compiled to the following:

```
#board .post {  
  border-radius: 0;  
  padding-bottom: 32px;  
}  
  
#board .reply {  
  padding: 0px 19px 0px 40px;  
}
```



Note that the `grunt-contrib-watch` daemon task recompiles `main.css` when `_bbs.scss` is saved.



4. Traverse to `src/coffee/app` and open `main.coffee`. This one file will implement the functionality for our BBS. The file consists of the following code:

```
$ ->
modalPost = $("#modal-post")
showModal = (formId, type) ->
  modalPost.find("#form-id").val formId
  modalPost.find("#form-message").val ""
  modalPost.find("#modal-type").html type
  modalPost.modal "show"

$("#nav-publish").click ->
  showModal "", "Post"

$(document).on "click", "a.btn-reply", ->
  showModal $(this).data("id"), "Reply"

modalPost.find("#btn-publish").click ->
  postId = modalPost.find("#form-id").val()
  message = $("#form-message").val()

  if typeof postId isnt "undefined" and
    not isNaN(parseInt(postId))
    $(".post[data-id=\"#{postId}\"]")
      .append("<hr />")
      .append "<div class=\"reply\">#{message}</div>"
  else
    id = new Date().getTime()
    board = $("#board")

    board.find("h1").remove()
    board.append "<div class=\"post well\">" +
      "data-id=\"#{id}\">#{message}" +
      "<a class=\"btn btn-primary btn-reply pull-right\">" +
      "data-id=\"#{id}\">Reply</a></div>"

  modalPost.modal "hide"
```



Note that the CoffeeScript keyword `isnt` translates to the typesafe `!==` operator in JavaScript instead of the looser `!=` operator.

For this project, CoffeeScript is kept simple for readability. The `showModal` function is used to prompt a Bootstrap modal to publish posts and replies. The click event bound to the `Publish Message` button renders posts and replies.

5. Lastly, traverse to `src/templates`. Open up `index.jade` and add the following script by replacing `// endbuild` at the bottom of the file with the following code:

```
script(type='text/javascript', src='scripts/app/main.js')
// endbuild
```

This `main.js` JavaScript file will be compiled once the associated CoffeeScript file `main.coffee` compiles.

Writing Mocha tests using Zombie and Assert

To automate the testing for our BBS, Mocha tests must be written first. These tests will involve short user stories, describing the behaviors being tested. CoffeeScript will be used to write these tests. Perform the following steps:

1. First, a test will be written to ensure that the initial page load yields the expected result. Take a look at the following block of code:

```
it "contains an empty #board div on startup", (done) ->
  done()
```

Replace the preceding block of code with the following:

```
it "contains an empty #board div on startup", (done) ->
  browser.visit "http://localhost:9001", ->
    assert.ok browser.success

    board = browser.query("#board")
    assert.ok board
    assert.equal browser.text("#board h1"), "No Posts
Found"

    done()
```

This test will visit `http://localhost:9001`, expecting a web page. The `grunt-contrib-connect` plugin will be used to accomplish this in the next section. It will assert that the only text found on the page is `No Posts Found`.

- Next, it is important to test whether or not the modal prompts on publish as required. Take a look at the following block of code:

```
it "prompts a #modal-post when clicking #nav-publish", (done) ->
  done()
```

Replace the preceding block of code with the following:

```
it "prompts a #modal-post when clicking #nav-publish", (done) ->
  assert.ok browser.success

  assert.equal browser.text("#modal-type"), ""
  modal = browser.query("#modal-post")
  browser.clickLink "#nav-publish", ->
    assert.equal browser.text("#modal-type"), "Post"
  done()
```

This test clicks the Publish a Post button in the navigation bar and asserts whether or not the hidden #modal-type field is Post, as required.

- Now that a test to prompt a modal has been created, the next test should check if publishing a post works as expected. Take a look at the following block of code:

```
it "publishes a .post when clicking #btn-publish", (done)
->
  done()
```

Replace the preceding block of code with the following:

```
it "publishes a .post when clicking #btn-publish", (done)
->
  assert.ok browser.query("#board h1")
  browser.fill("#form-message", "Here's a Post")
    .clickLink "#btn-publish", ->
    assert.ok not browser.query("#board h1")
    assert.equal browser.text("#board:nth-child(1)
span"),
  "Here's a Post"
  done()
```

This tests the publishing of a post. The message field is populated and the Publish Message button is pressed, asserting that the post is rendered as required.

4. After testing, if publishing a post works, the next logical move is to test if publishing a reply returns the expected rule. Take a look at the following block of code:

```
it "publishes a .reply when clicking #btn-publish",  
(done) ->  
  done()
```

Replace the preceding block of code with the following:

```
it "publishes a .reply when clicking #btn-publish", (done) ->  
  browser.clickLink "Reply", ->  
    assert.equal browser.text("#modal-type"), "Reply"  
  
    browser.fill("#form-message", "Here's a Reply")  
      .clickLink "#btn-publish", ->  
        assert.equal browser.text(".reply span"),  
          "Here's a Reply"  
        done()
```

The final test clicks on the Reply button, prompting a modal. The message field is then populated once again and the test asserts that a reply is populated as required.

Implementing the custom test Grunt task

In order to automate the testing process, it is important to set up the necessary Grunt components required for Mocha. Now, perform the following steps:

1. Open the `Gruntfile.js` file.
2. As shown in the previous section, the Mocha tests expect the website to be mounted on localhost, specifically on port 9001. To accomplish this, the `grunt-contrib-connect` plugin will be configured. Add the following lines of code after the initialization of the `config` object:

```
config['connect'] = {  
  server: {  
    options: {  
      port: 9001,  
      base: 'src'  
    }  
  }  
};
```

3. Next, the `grunt-mocha-test` plugin needs to know where the test directory is. Add the following code block after the initialization of the `config` object:

```
config['mochaTest'] = {  
  src: ['test/bbs.coffee']  
};
```

4. It is important to define the test task series now. Take a look at the following line of code:

```
var testTasks = [];
```

Replace the preceding block of code with the following:

```
var testTasks = [  
  'connect',  
  'mochaTest'  
];
```

5. Now, you can open up a terminal and run the automated test suite by issuing the following command:

```
grunt test
```

Implementing the custom build Grunt task

When deploying to production, a custom build Grunt task must be defined to scaffold the project together and strip away boilerplate. Set this up by performing the following steps:

1. Open the `Gruntfile.js` file.

Initially, the `grunt-contrib-clean` task will be reused as before. You should then add the following lines of code after the initialization of the `config` object:

```
config['clean'] = {  
  build: {  
    files: [{  
      dot: true,  
      src: [  
        'dist/*',  
        '!dist/.git*' ]  
    }]  
  }  
};
```

2. Like before, `grunt-usemin` will be used for this final project. It will help to minify and concatenate all JavaScript files involved with this project.

Add the following block to the Gruntfile:

```
config['useminPrepare'] = {
  options: {
    dest: 'dist'
  },
  html: 'src/index.html'
};

config['usemin'] = {
  options: {
    dirs: ['dist']
  },
  html: ['dist/{,*/}*.html']
};
```

3. Next, the `grunt-htmlmin-contrib` task will be set up. You should add the following code block to your Gruntfile:

```
config['htmlmin'] = {
  dist: {
    options: {
      collapseBooleanAttributes: true,
      removeAttributeQuotes: true,
      removeRedundantAttributes: true,
      removeEmptyAttributes: true
    },
    files: [{
      expand: true,
      cwd: 'src',
      src: '{,*/}*.html',
      dest: 'dist'
    }]
  }
};
```

Again, similar to the previous chapter, this code block will minify the amount of HTML that is required.

4. As required by `grunt-usemin`, the `grunt-contrib-uglify` task will be defined in this step. You should add the following block to your Gruntfile:

```
config['uglify'] = {
  options: {
    mangle: false
  }
};
```

5. The `grunt-contrib-copy` task will need to be defined for the last stage of the build task. You can achieve this by adding the following block to your Gruntfile:

```
config['copy'] = {
  build: {
    files: [{
      expand: true,
      dot: true,
      cwd: 'src',
      dest: 'dist',
      src: []
    }]
  }
};
```

6. In order to avoid conflict with the browser cache, the compiled files used for this project will need to have a cache bust prepended to their file names. This can be done by again adding the following block to your Gruntfile:

```
config['rev'] = {
  dist: {
    files: {
      src: [
        'dist/scripts/{,*/}*.js',
      ]
    }
  }
};
```

7. Finally, it is important to configure your `grunt-contrib-cssmin` task by inserting the following section of code:

```
config['cssmin'] = {
  dist: {
    files: {
      'dist/styles/main.css': [
        'src/styles/{,*/}*.css'
      ]
    }
  }
};
```

8. This is concluded by defining the build task. Take a look at the initialization of the tasks array that contains:

```
var buildTasks = [];
```

Replace the preceding code with the following block of code:

```
var buildTasks = [  
  'test',  
  'clean:dist',  
  'jade:dist',  
  'coffee:dist',  
  'compass:dist',  
  'useminPrepare',  
  'concat',  
  'uglify',  
  'copy',  
  'rev',  
  'usemin'  
];
```

What will happen now is that this task order will test the application, clean the `dist` directory, compile all Jade, CoffeeScript, and Sass files, and minify the generated HTML files.

9. Now, you can open up a terminal and finally scaffold your project by issuing the following command:

```
grunt build
```

Following this, you can open the website by running `dist/index.html` in your web browser.

Summary

This chapter covered the use of the Mocha testing engine alongside the `grunt-mocha-test` plugin. The `grunt-contrib-connect` plugin was used to host a standalone web server for the purpose of testing. All in all, the concepts covered in all the previous chapters came together in this final project, demonstrating the productivity gains found in using Grunt. The next chapter will cover the best practices found in modern web development, including form validation, responsive design, and page-speed optimization.

5

Best Practices for Modern Web Applications

This chapter will cover the best practices for frontend development today. It will cover load time reduction, search engine optimization, form validation, and responsive design.

The importance of search engine optimization

Every day, web crawlers scrape the Internet for updates on new content to update their associated search engines. People's immediate reaction to finding web pages is to load a query on a search engine and select the first few results. Search engine optimization is a set of practices used to maintain and improve search result ranks over time.

Item 1 – using keywords effectively

In order to provide information to web crawlers, websites provide keywords in their HTML meta tags and content. The optimal procedure to attain effective keyword usage is to:

- Come up with a set of keywords that are pertinent to your topic
- Research common search keywords related to your website
- Take an intersection of these two sets of keywords and preemptively use them across the website

Once this final set of keywords is determined, it is important to spread them across your website's content whenever possible. For instance, a ski resort in California should ensure that their website includes terms such as California, skiing, snowboarding, and rentals. These are all terms that individuals would look up via a search engine when they are interested in a weekend at a ski resort. Contrary to popular belief, the keywords meta tag does not create any value for site owners as many search engines consider it a deprecated index for search relevance. The reasoning behind this goes back many years to when many websites would clutter their keywords meta tag with irrelevant filler words to bait users into visiting their sites. Today, many of the top search engine have decided that content is a much more powerful indicator for search relevance and have concentrated on this instead.

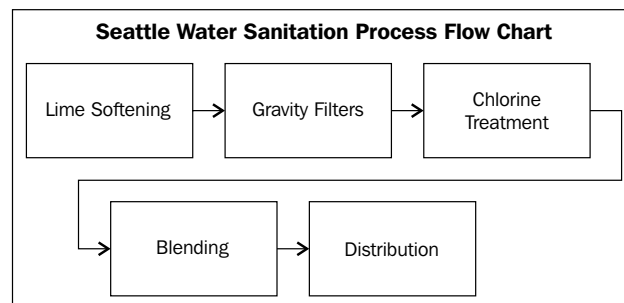
However, other meta tags, such as description, are still being used for displaying website content on search rankings. These should be brief but powerful passages to pull in users from the search page to your website.

Item 2 – header tags are powerful

Header tags (also known as h-tags) are often used by web crawlers to determine the main topic of a given web page or section. It is often recommended to use only one set of h1 tags to identify the primary purpose of the web page, and any number of the other header tags (h2, h3, and so on) to identify section headings.

Item 3 – make sure to have alternative attributes for images

Despite the recent advance in image recognition technology, web crawlers do not possess the resources necessary for parsing images for content through the Internet today. As a result, it is advisable to leave an `alt` attribute for search engines to parse while they scrape your web page. For instance, let us suppose you were the webmaster of **Seattle Water Sanitation Plant** and wished to upload the following image to your website:



Since web crawlers make use of the `alt` tag while sifting through images, you would ideally upload the preceding image using the following code:

```

```

This will leave the content in the form of a keyword or phrase that can help contribute to the relevancy of your web page on search results.

Item 4 – enforcing clean URLs

While creating web pages, you'll often find the need to identify them with a URL ID. The simplest way often is to use a number or symbol that maps to your data for simple information retrieval. The problem with this is that a number or symbol does not help to identify the content for web crawlers or your end users.

The solution to this is to use clean URLs. By adding a topic name or phrase into the URL, you give web crawlers more keywords to index off. Additionally, end users who receive the link will be given the opportunity to evaluate the content with more information since they know the topic discussed in the web page. A simple way to integrate clean URLs while retaining the number or symbol identifier is to append a readable **slug**, which describes the topic, to the end of the clean URL and after the identifier. Then, apply a regular expression to parse out the identifier for your own use; for instance, take a look at the following sample URL:

```
http://www.example.com/post/24/golden-dragon-review
```

The number 24, when parsed out, helps your server easily identify the blog post in question. The slug, `golden-dragon-review`, communicates the topic at hand to both web crawlers and users.



While creating the slug, the best practice is often to remove all non-alphanumeric characters and replace all spaces with dashes. Contractions such as `can't`, `don't`, or `won't`, can be replaced by `cant`, `dont`, or `wont` because search engines can easily infer their intended meaning. It is important to also realize that spaces should not be replaced by underscores as they are not interpreted appropriately by web crawlers.

Item 5 – backlink whenever safe and possible

Search rankings are influenced by your website's clout throughout websites that search engines deem as trustworthy. For instance, due to the restrictive access of .edu or .gov domains, websites that use these domains are deemed trustworthy and given a higher level of authority when it comes down to search rankings. This means that any websites that are backlinked on trustworthy websites are seen at a higher value as a result.

Thus, it is important to often consider backlinking on relevant websites where users would actively be interested in the content. If you choose to backlink irrelevantly, there are often consequences that you'll face, as this practice can often be caught automatically by web crawlers while comparing the keywords between your link and the backlink host.

Item 6 – handling HTTP status codes properly

Server errors help the client and server communicate the status of page requests in a clean and consistent manner. The following chart will review the most important server errors and what they do:

Status Code	Alias	Effect on SEO
200	Success	This loads the page and the content is contributed to SEO
301	Permanent redirect	This redirects the page and the redirected content is contributed to SEO
302	Temporary redirect	This redirects the page and the redirected content doesn't contribute to SEO
404	Client error (not found)	This loads the page and the content does not contribute to SEO
500	Server error	This will not load the page and there is no content to contribute to SEO

In an ideal world, all pages would return the 200 status code. Unfortunately, URLs get misspelled, servers throw exceptions, and old pages get moved, which leads to the need for other status codes. Thus, it is important that each situation be handled to maximize communication to both web crawlers and users and minimize damage to one's search ranking.

When a URL gets misspelled, it is important to provide a 301 redirect to a close match or another popular web page. This can be accomplished by using a clean URL (referenced in the *Item 4 – enforcing clean URLs* section of this chapter) and parsing out an identifier, regardless of the slug that follows it. This way, there exists content that contributes directly to the search ranking instead of just leaving a 404 page.

Server errors should be handled as soon as possible. When a page does not load, it harms the experience for both users and web crawlers, and over an extended period of time, can expire that page's rank.

Lastly, the 404 pages should be developed with your users in mind. When you choose not to redirect them to the most relevant link, it is important to either pass in suggested web pages or a search menu to keep them engaged with your content.

The `connect-rest-test` Grunt plugin can be a healthy addition to any software project to test the status codes and responses from a RESTful API. You can find it at <https://www.npmjs.org/package/connect-rest-test>.

Alternatively, while testing pages outside of your RESTful API, you may be interested in considering `grunt-http-verify` to ensure that status codes are returned properly. You can find it at <https://www.npmjs.org/package/grunt-http-verify>.

Item 7 – making use of your robots.txt and site map files

Often, there exist directories in a website that are available to the public but should not be indexed by a search engine. The `robots.txt` file, when placed in your website's root, helps to define exclusion rules for web crawling and prevent a user-defined set of search engines from entering certain directories.

For instance, the following example disallows all search engines that choose to parse your `robots.txt` file from visiting the music directory on a website:

```
User-agent: *  
Disallow: /music/
```

While writing navigation tools with dynamic content such as JavaScript libraries or Adobe Flash widgets, it's important to understand that web crawlers have limited capability in scraping these. Site maps help to define the relational mapping between web pages when crawlers cannot heuristically infer it themselves. On the other hand, the `robots.txt` file defines a set of search engine exclusion rules, and the `sitemap.xml` file, also located in a website's root, helps to define a set of search engine inclusion rules. The following XML snippet is a brief example of a site map that defines the attributes:

```
<?xml version="1.0" encoding="utf-8"?>

<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
  <url>
    <loc>http://example.com/</loc>
    <lastmod>2014-11-24</lastmod>
    <changefreq>always</changefreq>
    <priority>0.8</priority>
  </url>
  <url>
    <loc>http://example.com/post/24/golden-dragon-review</loc>
    <lastmod>2014-07-13</lastmod>
    <changefreq>never</changefreq>
    <priority>0.5</priority>
  </url>
</urlset>
```

The attributes mentioned in the preceding code are explained in the following table:

Attribute	Meaning
loc	This stands for the URL location to be crawled
lastmod	This indicates the date on which the web page was last modified
changefreq	This indicates the page is modified and the number of times the crawler should visit as a result
priority	This indicates the web page's priority in comparison to the other web pages

Using Grunt to reinforce SEO practices

With the rising popularity of client-side web applications, SEO practices are often not met when page links do not exist without JavaScript. Certain Grunt plugins provide a workaround for this by loading the web pages, waiting for an amount of time to allow the dynamic content to load, and taking an HTML snapshot. These snapshots are then provided to web crawlers for search engine purposes and the user-facing dynamic web applications are excluded from scraping completely.

Some examples of Grunt plugins that accomplish this need are:

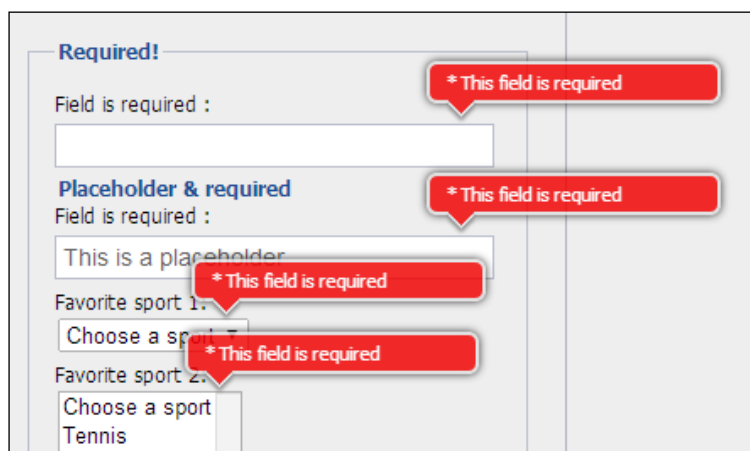
- `grunt-html-snapshots` (<https://www.npmjs.org/package/grunt-html-snapshots>)
- `grunt-ajax-seo` (<https://www.npmjs.org/package/grunt-ajax-seo>)

Form validation in the modern web world

The first task a user often needs to do is sign up in order to access the full content of a website. With the growing number of web applications in the world today, this process should be frictionless and simple, guiding users through it. With new client-side libraries used exclusively for form validation, this process can be accomplished in a much easier fashion than before.

Item 8 – using client-side validation over error pages

In the past, when a user submitted invalid form data, it would be validated on the server side, and the user was redirected to an error page. Nowadays, client-side libraries allow web developers to embed dynamic validation on the client side, sending inline errors preemptively to users to notify them when the data is missing or invalid. This integration helps to ensure that data is validated before sending it to the server, which prevents any friction from completing the sign up process once the user submits their information. One such example is shown in the following screenshot:



The screenshot displays a web form with several input fields. Each field has a red error message bubble next to it that reads "* This field is required". The fields and their corresponding error messages are:

- Required!**: A text input field with the error message "* This field is required".
- Placeholder & required**: A text input field with the placeholder text "This is a placeholder" and the error message "* This field is required".
- Favorite sport 1.**: A dropdown menu with the selected option "Choose a sport" and the error message "* This field is required".
- Favorite sport 2.**: A dropdown menu with the selected option "Choose a sport" and the error message "* This field is required".

The form also includes a "Tennis" option in the dropdown menu for "Favorite sport 2."

The use of client-side form validation libraries easily improves one's user experience.

Item 9 – differentiating required and optional information

Occasionally, there will be inputs that do not store the required information. Examples include personal details such as phone numbers or postal codes or extraneous user information such as biographies. It is important to note that although this information is not required to be provided, it still can add unnecessary friction to your user's experience. Thus, it is worthwhile to evaluate whether or not this information even adds value to your experience as a product owner.

While listing required fields, it is important to label them with an identifier such as a red asterisk (*). A client-side library can then be used to validate that the field is filled out and is valid.

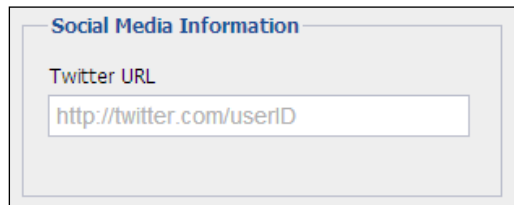
Item 10 – avoiding confusing fields

Certain fields are more complex than others when it comes down to validity. Thus, it is important to consider the steps that a user must take while filling out a field.

For instance, consider a field that requires a user's Twitter ID. The possible questions include:

- What does an ID look like?
- What constitutes a valid ID?
- What steps does a user need to take to find and copy their ID in?
- Does it need to be prefixed with a @ character, just like it's done on Twitter?

An easier approach to enter data to this field would be to request the user's Twitter URL instead, as shown in the following screenshot. This way, all they need to do is visit their Twitter feed, copy the associated URL, and paste it into the field. By adding an example URL into the input as a placeholder, you can help guide the user into understanding what is required.



The screenshot shows a form titled "Social Media Information" in blue text. Below the title is a label "Twitter URL" in red text. Underneath the label is a text input field with a light gray border. Inside the input field, the text "http://twitter.com/userID" is displayed in a light gray font, serving as a placeholder for the user's input.

Using placeholders in input boxes helps users identify the format for which their data should be input.

Lastly, as a developer, it is easy to fall into the trap of defining a class and populating a form with fields that represents the members of that class. It is important, however, to sit back and review the fields and figure out which ones need clarification for users who intend to use your form. Add the explanations, as necessary, to your inputs by placing them either on the side or as a tooltip.

Item 11 – using confirmation fields for pertinent data

While filling out their forms, users may often misspell their information. Certain data, such as passwords, cell phone numbers, or e-mails, may be required for the sign-up or sign-in process and must be correct. The best practice for validating this is often asking the user to type in their data twice, in two adjacent fields, to ensure that their information was not mistyped.



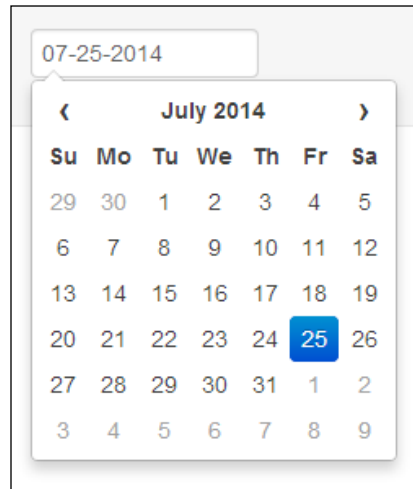
It is worth considering whether or not a field requires an adjacent confirmation partner. If the data can be changed upon login, the user can always ensure that it is accurate in the future. E-mail addresses and cell phone numbers often only require confirmation fields when they are strictly mandatory for sign-up, such as when used for e-mail or SMS validation. This is, however, all up to the web developer's discretion.

Item 12 – using custom inputs for complex data types

When accepting strings or numbers in inputs, it is fairly simple to validate and accept the data as is for use. For other data types, such as dates, URLs, or hexadecimal color codes, it is usually worthwhile to incorporate custom inputs via JavaScript libraries or HTML5 features to force a specific format.

For instance, it is possible for you to specify that an input must be in the format of MM/DD/YY while taking in a date. However, a user may accidentally disregard the format requirement and type it in as DD/MM/YY instead. If the date was March 4, 2014, both formats would pass a server-side validation.

While using a custom calendar input, the user simply needs to select the date on the widget, which alleviates the risk of false information. A great example of this plugin is the `bootstrap-datepicker` plugin, as shown in the following screenshot:



It can be found at <https://github.com/eternicode/bootstrap-datepicker>.

Item 13 – preventing autovalidation with CAPTCHAs

When running a website that provides a service, bots may be written by third parties to take advantage of it. These bots often automate the registration of accounts and website processes in order to phish user details, scrape website content, or take part in online polls.

Thankfully, this practice can be halted with the use of **CAPTCHA**. Although imperfect, CAPTCHAs can act as a significant enough barrier of entry for most bot scripters. By incorporating a modern CAPTCHA library such as reCAPTCHA, as shown in the following screenshot, web developers are now able to prevent the vast majority of bots from abusing their services:



More information can be found on reCAPTCHA's official website at [recaptcha.net](https://www.recaptcha.net).

Item 14 – reinforcing data integrity with server-side validation

While accepting information through the client, it is important to ensure that the data is valid on the server side. A client-side form validation library may fail to enforce validity because it didn't load or had a premature exit. Additionally, hackers may choose to work around the client-side enforcement, by sending data as a direct HTTP request.

Server-side validation should have at least the same standards as the client-side one. Additionally, server-side validation should enforce type safety, ensuring that all number fields are indeed numbers, string fields are strings, and so forth. If possible, it is important to ensure that postal codes, phone numbers, and other pertinent data are real on the server side by conducting a registry lookup.

Using Grunt to automate form testing

Earlier, in *Chapter 4, Final Project – Simple Bulletin Board System*, the use of `grunt-contrib-mocha` alongside the headless web client, *Zombie*, was discussed. This `grunt-contrib-mocha` plugin can easily be used to test both client and server validation through filling out forms and issuing direct HTTP requests. Along with the standard assertion library, this set of tools can test the following conditions:

- Whether the client-side form validation library loads and operates correctly
- Whether the required fields react when unfilled
- Whether the fields with a specific format react when invalid
- Whether the server-side validation returns the same result as the client-side configuration would

Designing interfaces for the mobile generation

As more people are adopting mobile technology in the form of phones and tablets, website designers have begun to take on new mediums in design and are changing their layouts and content accordingly. Presenting a redirect to the download link of a native mobile application is no longer the best practice, as users shift towards preferring the mobile web. Thus, it is ideal to follow the best practices involved with responsive web design throughout one's website. To accomplish this, many choose to use new frameworks such as Twitter, Bootstrap, or ZURB Foundation. This section will discuss these frameworks in more detail.

Item 15 – designing preemptively with mobile in mind

Unlike standard web design, responsive layouts must be implemented with multiple devices in mind. Thus, one cannot simply convert a simple web design to a responsive one without aggressively conducting changes.

Alternatively, a web designer may opt to display completely different designs across platforms using CSS3 media queries and breakpoints, as shown in the following code:

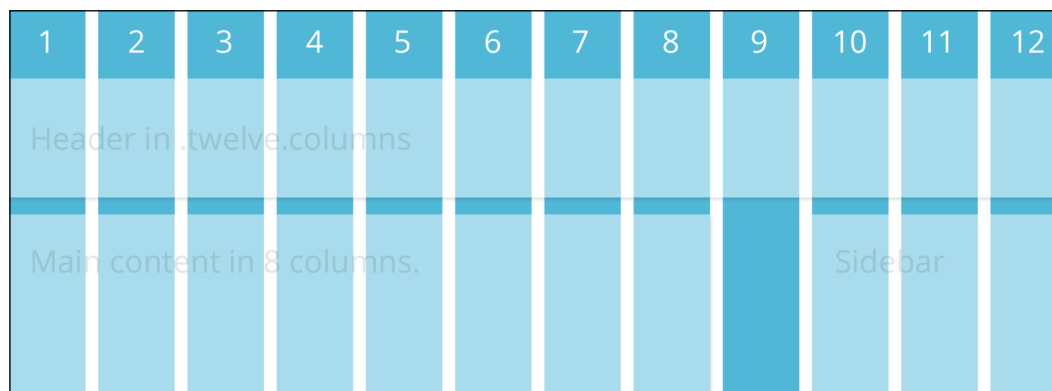
```
@media (max-width:767px) {  
  .visible-mobile {  
    // Styles for Smartphones and Smaller Tablets  
  }  
}  
  
@media (min-width:768px) {  
  .visible-desktop {  
    // Styles for Larger Tablets and Desktops  
  }  
}
```

In this case, the `.visible-mobile` and `.visible-desktop` classes help show the content based on the device that renders it. The following table shows other breakpoints of interest and their associated screen sizes:

Breakpoint	Screen sizes
< 480 px	Smaller smartphones
< 768 px	Small tablets and larger smartphones
> 768 px	Larger tablets and desktops
< 320 px	Older, low-resolution phones
> 1024 px	Wide desktops

It is important to note that mobile devices tend to load and render web content at a much slower pace than desktop computers. Additionally, larger mobile devices such as tablets often have a higher computing power than smartphones and are therefore capable of loading more content when required.

Responsive design, although conceptually simple, can be tedious to implement manually. External frameworks such as Twitter Bootstrap or ZURB Foundation have provided a multitude of CSS classes and JavaScript libraries for responsive design. Bootstrap 3 and Foundation both adopt a 12-column grid system, as shown in the following screenshot, which allows web designers to adjust content containers based on the device used to render it:



The 12-column grid system, found in both Foundation and Bootstrap, helps designers allocate space in a systematic fashion on desktop and mobile devices alike.

These frameworks can be found in the following links:

- Twitter Bootstrap (<http://getbootstrap.com/>)
- ZURB Foundation (<http://foundation.zurb.com/>)

This information, along with the use of responsive design, should demonstrate the importance of distributing separate content and designs to different platforms. For instance, when strategizing what to place on a smartphone-rendered web design, it is important to convey the same message that would be presented on tablet or desktop devices but with less images and text. This way, users are able to enjoy a clean but effective experience that they would otherwise receive on a larger device.

Item 16 – lazy load content using JavaScript

When loading content for users, it is a common practice to load everything the user will ever need to see for a particular web page. It is possible to lazy load content through the use of streams, only loading images or files when a user has hit a certain area of the web page.

An example of this is a clothing website. On a mobile device, a web designer may decide to load all of the articles of clothing at once. Alternatively, he may choose to load the items as the user scrolls down on either a phone or tablet, loading them on-the-fly. This way, the initial page load of the web page is drastically reduced. It can be argued, however, that the lazy loading of content actually increases the overall page load time if used excessively, as it increases the amount of HTTP requests required. This method, however, is ubiquitous across so many mainstream apps that users expect such behavior. Thus, it can be argued to be an effective best practice.

Item 17 – defer parsing of JavaScript

When JavaScript is loaded through a web browser, it blocks all other resources from getting loaded until it is processed. Thus, JavaScript files are often considered the critical path for loading pages. There are simple ways to combat this such as appending JavaScript inclusion tags at the bottom of your HTML files. Unfortunately, these methods do not work well on mobile devices that are not yet optimized for JavaScript processing.

Thus, a solution that many web developers have adopted is deferring the processing of JavaScript. To accomplish this, you can include the JavaScript file by embedding it directly into the DOM within a script tag. Many libraries, such as `deferred`, `Q`, and `bluebird` help to enable the deferred JavaScript. You can find out more about them via the following links:

- `deferred` (<https://github.com/medikoo/deferred>)
- `Q` (<https://github.com/krisKowal/q>)
- `bluebird` (<https://github.com/petkaantonov/bluebird>)

Using Grunt to reduce page load time

As with the concepts found in this book, concatenating CSS and JavaScript files using Grunt helps to reduce HTTP requests, increasing the page load speed. Furthermore, minifying the contents of these concatenated files is another option Grunt provides you with, reducing the file size of the contents being transferred across the network. Lastly, the use of `grunt-rev`, as demonstrated throughout the book, allows you to safely append a revision hash to your files, purging the cache that your users' web browsers stored for files. This way, you can safely allow clients to cache CSS, JavaScript, or image files, reducing page load time.

Summary

Over the years, the modern Web world and its best practices have evolved as users expected a better experience over time. As a result, tools such as Grunt have become robust and are available to developers around the world, which allows them to build their web applications in an automated fashion. Grunt and its various plugins provide immense functionality for the realms of search engine optimization, page load time, and user experience design, which presents strong arguments for its tool chain. This chapter should allow you to fully understand the importance of Grunt in web development, yielding great gains over time.

From this point onward, it is recommended that readers look into the tools used in this book more deeply. As tools such as CoffeeScript, Sass and testing practices change, so will the considerations made while configuring their respective Grunt plugins. It is advisable for readers to also keep up with existing and future best practices on the Web, in order to grow as developers in the modern world.

Index

A

autovalidation

preventing, with CAPTCHAs 84

B

backlink 78

BBS. See **Simple Bulletin Board System**

blog

building 27

custom build Grunt task,
implementing 40-42

developing 32-40

grunt-contrib-watch, configuring 28-32

Grunt plugins, installing 27, 28

bluebird

URL 88

Bower

about 16

basics 18

installing 16

logo 16

using 16

C

CAPTCHAs

autovalidation, preventing 84

cascading style sheet (CSS) 26

client-side validation

using, over error pages 81

CoffeeScript 45

CommonJS spec 7

contrib packages 6

cssDir variable 30

custom build Grunt task

implementing 40-74

custom test Grunt task

implementing 70

D

deferred

URL 88

dist 23

E

employee management system

building 46

custom build Grunt task,
implementing 56-59

developing 49-56

grunt-contrib-watch, configuring 47-49

Grunt plugins, installing 46, 47

F

**fatal error: Unable to find local
grunt** 19

form validation

about 81

autovalidation, preventing with
CAPTCHAs 84

client-side validation, using over error
pages 81

confirmation fields, using 83

confusing fields, avoiding 82, 83

custom inputs, using for complex data
types 83, 84

required and optional information,
 differentiating 82
server-side validation 85
test automating, Grunt used 85

G

Git

about 8
basics 10
installing 9
installing, on Linux 9
installing, on Mac OS X 9
installing, on Windows 9
using 9

GitHub

about 11
installing 12, 13
logo 11
using 11

Grunt

about 5, 6
installing 18, 19
logo 6
troubleshooting 19

Grunt: command not found error 19

grunt-contrib-compass task 29

grunt-contrib-jade plugin 31

grunt-contrib-watch
 configuring 28-49

Gruntfile.js configuration file

configuration section 7
constants 7
dissecting 7
functions 7
user-defined tasks 8

Grunt plugins

installing 27, 28

grunt-usemin plugin 56

grunt-usemin task 56

H

header tags 76

Hello World page

 deploying 20-23

h-tags. *See* header tags

HTTP status codes

 handling 78, 79

I

importPath variable 30

installation

Bower 16
GitHub 12, 13
Git, on Linux 9
Git, on Mac OS X 9
Git, on Windows 9
Grunt 19
Grunt plugins 27, 28
npm 14
npm, on Linux 14
npm, on Mac OS X 14
npm, on Windows 14

J

Jade

about 25
logo 25, 26

L

Linux

Git installation 9
npm installation 14

M

Mac OS X

Git installation 9
npm installation 14

mobile generation

interfaces, designing 85-87
JavaScript parsing, deferring 88
lazy load content, JavaScript used 87, 88
page load time reducing, Grunt used 88

Mocha

- about 61
- grunt-contrib-watch, configuring 63-65
- Grunt plugins, installing 62, 63
- logo 61

Mocha tests

- writing 68-70

N

Node Package Manager. See npm

npm

- about 13
- installing 14
- installing, on Linux 14, 15
- installing, on Mac OS X 14, 15
- installing, on Windows 14
- using 14, 15

P

page load time

- reducing, Grunt used 88

Q

- URL 88

R

robots.txt file 79, 80

S

Sass

- about 26
- logo 26

search engine optimization

- backlinks 78
- clean URLs, enforcing 77
- effective keyword usage 75, 76
- header tags 76
- HTTP status codes, handling 78, 79
- images, alternative attributes 76, 77
- importance 75
- reinforcing, Grunt used 80
- robots.txt, using 79, 80
- site map files, using 79, 80

server-side validation 85

Simple Bulletin Board System

- custom build Grunt task, implementing 71-74
- custom test Grunt task, implementing 70
- developing 65-68
- tests, automating 68-70

site map file 79, 80

slug 77

T

troubleshooting

- fatal error: Unable to find local grunt 19
- Grunt: command not found error 19

Twitter Bootstrap

- URL 87

U

user-defined tasks 8

W

Windows

- Git installation 9

Z

ZURB Foundation

- URL 87



Thank you for buying Mastering Grunt

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, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: 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 licences, 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, 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.

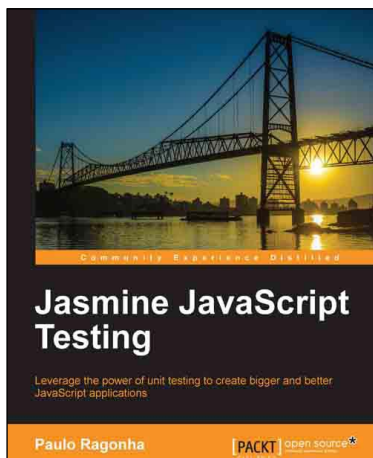


Getting Started with Grunt: The JavaScript Task Runner

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.



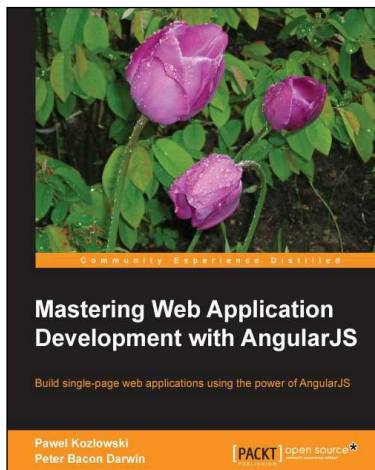
Jasmine JavaScript Testing

ISBN: 978-1-78216-720-4 Paperback: 146 pages

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

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

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



Mastering Web Application Development with AngularJS

ISBN: 978-1-78216-182-0 Paperback: 372 pages

Build single-page web applications using the power of AngularJS

1. Make the most out of AngularJS by understanding the AngularJS philosophy and applying it to real life development tasks.
2. Effectively structure, write, test, and finally deploy your application.
3. Add security and optimization features to your AngularJS applications.
4. Harness the full power of AngularJS by creating your own directives.



SproutCore Web Application Development

ISBN: 978-1-84951-770-6 Paperback: 194 pages

Creating fast, powerful, and feature-rich web applications using the SproutCore HTML5 framework

1. Write next-gen HTML5 apps using the SproutCore framework and tools.
2. Get started right away by creating a powerful application in the very first chapter.
3. Build your understanding of SproutCore as you follow through the most complete reference to the framework anywhere in existence.

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