Quick answers to common problems

Grunt Cookbook

Over 80 hands-on recipes for streamlining development, management, and deployment with Grunt

Jurie-Jan Botha

[PACKT] open source*
PUBLISHING
community experience distilled

Grunt Cookbook

Table of Contents

Grunt Cookbook

Grunt Cookbook

First published: June 2015

Production reference: 1250615

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-78328-651-5

www.packtpub.com

Credits

Author

Jurie-Jan Botha

Reviewers

Alejandro Hernández

Mark McDonnell

Volodymyr Tsvang

Commissioning Editor

Sam Birch

Acquisition Editor

Shaon Basu

Content Development Editor

Athira Laji

Technical Editor

Madhunikita Sunil Chindarkar

Copy Editor

Adithi Shetty

Project Coordinator

Harshal Ved

Proofreader

Safis Editing

Indexer

Hemangini Bari

Production Coordinator

Shantanu N. Zagade

Cover Work

Shantanu N. Zagade

About the Author

Jurie-Jan Botha has been immersed in the computer programming world from a very early age. He taught himself by working his way through textbooks and later (when it became available) through the Internet. Being involved in a variety of projects, such as games, content management, and telecoms systems, throughout his career has provided him with a fairly good insight into selecting the right tools for the job and making optimal use of them. He's been using Grunt extensively for the past 3 years and is looking forward to using it in the foreseeable future.

I would like to thank all my family, friends, and colleagues throughout the years, and more importantly, all those who contributed to the various projects in the open source community.

About the Reviewers

Alejandro Hernández (aka picanteverde) is a full-stack JavaScript engineer (and JavaScript fanatic) devoted to spreading the use of JavaScript in ingenious ways. A JavaScript developer since 1997, Alejandro has worked on several JavaScript and web-related projects over the course of his career, learning and playing with new technologies (HTML5, Node.js, and ECMAScript 5, 6, and 7) as soon as they became available. Now he enjoys doing research on multiple browsers' interaction patterns using HTML5 features.

Alejandro worked in the research department for major companies such as Intel and Globant. Currently, he is a JavaScript specialist at Toptal LLC and helps companies take platform and architecture decisions in web environments in general.

I would like to thank my dear girlfriend, Carlita, whose support has always been my source of strength and inspiration; my family who have always been there for me whenever I needed them; and my friend Mauricio, who is always ready for a deep and complicated technical discussion on JavaScript.

Mark McDonnell currently works at the BBC as a senior engineer for the Responsive News project. He was originally hired as a frontend specialist focusing on the three tenets of web development: JavaScript, CSS, and HTML. Mark has moved further up the stack and can now be found working on applications in JRuby and PHP while shell scripting his way around multiple Jenkin CI servers and Vim/tmux'ing his way to the cloud utilizing the many AWS services via BBC's own internal abstraction layer: Cosmos.

Mark is an author of many online blog posts on Smashing Magazine, NetTuts, as well as his own website. He also has articles printed in the popular NET Magazine and is a published author with Apress with the title Pro Vim.

Along with honing his skills in classical object-oriented design and functional programming concepts from Lisp (in particular Clojure), Mark has aspirations of becoming a true polyglot developer one day.

You can find Mark online at the following locations:

- https://twitter.com/integralist
- https://github.com/integralist

- http://www.integralist.co.uk/

Volodymyr Tsvang is a software engineer with a primary focus on web development. He's spent the majority of his time working on frontend, as well as backend, JavaScript projects. Volodymyr has been using Grunt since its earliest release. He's also the creator and an active maintainer of the Grunt plugin to compile Dust.js templates. Since 2012, Volodymyr has been working as a JavaScript developer at Under Development LLC, which is one of the greatest Russian software development companies.

Preface

Grunt is a task automation tool that provides the platform for an ever-growing ecosystem of plugins. These plugins provide the actual implementation for the tasks that can be automated. File management, testing, deployment, template rendering, code generation, and much more can be automated using easily configurable tasks.

The Grunt platform's key strength lies in the simple methodology it has defined for task configuration, and the fact that all the plugins it supports tend to adhere to this methodology. This provides developers with a faster way to start using tools that would previously have taken more time to learn, while also presenting the configuration in a standard format that can be easily understood, manipulated, and shared.

The Grunt Cookbook aims to provide easy-to-follow instructions to automate all kinds of tasks that your projects need to perform on a regular basis. We'll start with the most basic and common tasks, work our way up to creating our own tasks, and eventually generate entire sites using only Grunt.

The chapters of this book will each focus on a specific topic for which a collection of recipes will be provided. Many of the recipes will explore the automation of similar tasks, using different tools or libraries. Each recipe also explores popular variations on its theme, just in case you have more specific needs.

What this book covers

Chapter 1, *Getting Started with Grunt*, starts with working though setting up Grunt to be used within a Node.js-based project, and covers some of the more common automations.

Chapter 2, *File Management*, covers the copying, compressing, linking, concatenation, and download of files. Working with file is probably the most common task that we'll encounter in any software project.

Chapter 3, *Templating Engines*, covers the rendering, compilation, and packaging of templates using some of the more popular engines. The practice of rendering content from templates is essential in the development of web-based projects.

Chapter 4, *Generating CSS and JavaScript*, covers the generation of CSS and JavaScript code. Using new languages that compile to JavaScript and CSS preprocessors that generate CSS can save time and improve the flexibility of a project.

Chapter 5, *Running Automated Tests*, covers the running of test suites and generating code coverage reports. Automated testing has become an essential part of all larger software projects and is an invaluable tool to ensure the stability and quality of code.

Chapter 6, *Deployment Preparations*, covers the optimization of images, minifying of CSS, ensuring the quality of our JavaScript code, compressing it, and packaging it all together into one source file.

Chapter 7, *Deploying to the End User*, covers the transfer of files to a network location, refreshing caching services and running commands on remote servers. Once we've got a functioning and optimized web application, it's time to make it accessible to its intended users.

Chapter 8, *Creating Custom Tasks*, covers all the aspects involved in creating our own custom tasks. At some point, you may encounter something that you'd like to automate but can't find a Grunt task that works exactly the way you need it to. This is when building a custom task can become invaluable to your operations and potentially make a hero out of you.

Chapter 9, *Authoring Plugins*, covers the process of discovering plugins, contributing to the existing plugin projects, and creating plugin projects of our own. The Grunt project would be nothing without its ecosystem of plugins, and we as developers, can partake in their creation and evolution.

Chapter 10, *Static Sites*, generates an entire static website using the Assemble plugin. Simpler websites that host content that doesn't change very often can mostly be generated once and uploaded to a hosting service.

What you need for this book

The only requirement for using the recipes in this book is a Node.js installation. This can be done either through downloading and running the installer, compiling from the source code, using a tool such as Node Version Manager (https://github.com/creationix/nvm), or using the package manager supported by your operating system.

If you have never worked with Grunt before, it's highly recommended that you start with Chapter 1, *Getting Started with Grunt*, as all the later chapters will reference it.

Who this book is for

This book will be useful to anyone who wishes to build anything from a static website to a more modern web application. Some basic JavaScript experience is preferable and some rudimentary knowledge of the Node.js platform could also come in handy.

Sections

In this book, you will find several headings that appear frequently (Getting ready, How to do it..., How it works..., There's more..., and See also).

To give clear instructions on how to complete a recipe, we use these sections as follows:

Getting ready

This section tells you what to expect in the recipe, and describes how to set up any software or any preliminary settings required for the recipe.

How to do it…

This section contains the steps required to follow the recipe.

How it works…

This section usually consists of a detailed explanation of what happened in the previous section.

There's more…

This section consists of additional information about the recipe in order to make the reader more knowledgeable about the recipe.

See also

This section provides helpful links to other useful information for the recipe.

Conventions

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

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "The simplest way to create a `package.json` file is to use the `npm init` command."

A block of code is set as follows:

```
{
  "name": "myproject",
  "version": "0.0.0",
  "description": "My first Grunt project.",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

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

```
'curl-dir': {
  weather: {
    router: function (url) {
      var city = url.slice(url.indexOf('=') + 1, url.length);
      return city + '.json';
    },
    src: 'http://api.openweathermap.org/'
      + 'data/2.5/weather?q={London,Paris,Tokyo}',
    dest: 'weather'
  }
}
```

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

```
Running "newer:jshint" (newer) task
Running "newer:jshint:sample" (newer) task
No newer files to process.
```

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

Note

Warnings or important notes appear in a box like this.

Tip

Tips and tricks appear like this.

Reader feedback

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

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

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

Customer support

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

Downloading the example code

Errata

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

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

Questions

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

Chapter 1. Getting Started with Grunt

In this chapter, we will cover the following recipes:

- Installing the Grunt CLI
- Installing Grunt on a project
- Installing a plugin
- Setting up a basic web server
- Watching files for changes
- Setting up LiveReload
- Processing only changed files
- Importing external data

Introduction

Grunt is a popular new task automation framework built upon the **Node.js** platform. It offers a wide range of features that allow you to streamline your project workflow and save time and energy by automating repetitive tasks, such as checking code quality, running tests, compiling templates and code, publishing to various types of services, and much more.

Task automation has been around since the beginning of software development and can probably be seen as a prominent reason for it being around at all. We're mostly writing programs to automate repetitive tasks after all.

Grunt itself is, for the most part, only a highly pluggable framework that provides a consistent interface for configuring automated tasks. The actual logic of the tasks is provided by a large variety of modules called **plugins**, which make use of this framework and usually tend to specialize in a certain set of functionalities.

At the time of writing the Grunt project is more than 3 years old, has over 3,000 plugins available in the **npm** public package registry, and provides tools and guides for creating or contributing to existing plugin projects.

A vast number of projects are currently making active use of Grunt in various ways, some of the most notable being the Yeoman, Modernizr, AngularJS, and JQuery projects.

Tip

Be sure to pay a visit to the Grunt website, if you've not already done so. It's filled with excellent guides and documentation, and it's the best place to find the plugins you need. The website can be found at the following URL:

http://gruntjs.com/

Installing the Grunt CLI

In order to make use of a Grunt configuration file, the Grunt **command-line interface** (**CLI**) tool needs to be installed.

Command-line tools such as the Grunt CLI are usually installed globally. This means that they are installed on top of the Node.js installation that is currently active in your terminal, and not in the current project path, as is usually the case.

Tip

In this book, we'll work with version 0.4.x of Grunt, which requires Node.js version 0.8.x or higher.

How to do it...

The following steps will take us through installing the Grunt CLI and testing for its successful installation.

1. Assuming that you already have a global installation of Node.js, the following is the command to install the Grunt CLI:

   ```
   $ npm install --global grunt-cli
   ```

2. If the installation was successful, the `grunt` command should now be available on the terminal. Test this by typing `grunt` in your terminal and confirm that it returns a message similar to the following:

   ```
   grunt-cli: The grunt command line interface. (v0.1.13)

   Fatal error: Unable to find local grunt.

   If you're seeing this message, either a Gruntfile wasn't found or
   grunt hasn't been installed locally to your project. For more
   information about installing and configuring grunt, please see
   the Getting Started guide:

   http://gruntjs.com/getting-started
   ```

How it works...

The `npm install` command looks up the `grunt-cli` package on npm's **public package registry**, and proceeds to download and install it once it is found.

Using the `-g` argument along with the `install` command indicates that the package we'd like to install, should be installed globally, meaning it should be installed on the version of Node.js that is currently active in our terminal.

In a default Node.js setup, a folder for executable binaries will automatically be added as a path that should be scanned by the terminal for executable commands. This makes the `grunt` command automatically available after the installation of this package, as its executable binary is provided and indicated in the package's installation information.

Setting up Grunt in a project

For a project to make use of the Grunt framework, it will require the installation of its libraries and the setting up a bare minimum configuration file. The libraries provide the framework and tools required by all Grunt plugins, and the configuration file provides a starting point from which we can start loading plugins and adjusting their behavior.

Getting ready

It's usually a good idea for a project to be packaged in a way to help keep track of dependencies, binaries, scripts, maintainers, and other important information. The standard package format for Node.js-based projects is **CommonJS**.

Tip

To find out more about CommonJS, you can take a look at its specification at the following URL:

http://wiki.commonjs.org/wiki/Packages/1.1

At the heart of the CommonJS package, lies the `package.json` file. This file contains everything important about the package and is stored in the JSON format. The simplest way to create a `package.json` file is to use the `npm init` command. This command will ask a series of questions and generate a `package.json` file based on the answers provided. Here's an example of the questions that are asked when you run the command:

```
name: (grunt-book) myproject
version: (0.0.0)
description: My first Grunt project.
entry point: (index.js)
test command:
git repository:
keywords:
author:
license: (ISC)
```

After these questions are answered, a `package.json` file will be generated in the current path with the following contents:

```
{
  "name": "myproject",
  "version": "0.0.0",
  "description": "My first Grunt project.",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

Tip

Another handy guide to the `package.json` file can be found at the following URL:

http://browsenpm.org/package.json

How to do it...

The following steps take us through installing the Grunt framework libraries on our project and creating a bare minimum configuration file.

1. First, we'll install the Grunt libraries in the current project path, and have it added to our project dependencies. This can all be done by using the following command:

   ```
   $ npm install --save grunt
   ```

   ### Tip

   Due to our use of the `--save` flag with the `install` command, the Grunt package will be added to the dependency list of the project's package. This can be confirmed by taking a look at the `dependencies` property inside the `package.json` file.

   The `--save-dev` flag is also available for use with the `install` command when you'd like the installed packages to be added to the `devDependencies` property, which lists the dependencies to set up a development environment.

2. Next, we'll set up an empty configuration file that would, at the very least, allow Grunt to run and also provides a place for future task configurations. Let's create a file called `Gruntfile.js` in the root directory of our project with the following contents:

   ```
   module.exports = function (grunt) {
     grunt.initConfig({});
     grunt.registerTask('default', []);
   };
   ```

3. Now that we have the Grunt libraries installed and a basic configuration file set up, we can use the `grunt` command to test that it's all working as expected. Running the `grunt` command in the terminal should now produce output similar to the following:

   ```
   Done, without errors.
   ```

   ### Tip

   Running the `grunt` command without specifying any parameters will always try to run the `default` task, which in the case of our current example, is set to nothing.

How it works...

When the Grunt CLI tool is used, it always looks for the nearest file named `Gruntfile.js`, from which it then attempts to load configurations. Inside the configuration file, there is an exported function that receives one argument. This argument is an object that provides us with access to the Grunt framework to load, create, and configure tasks.

At this point, we have no tasks loaded or created, and no configurations defined. Our `default` task is also set to do nothing, so running the `grunt` command did nothing except report that it was successfully completed.

Installing a plugin

All of the functionality that can be provided by Grunt is housed in the plugins that are made available in the form of Node.js packages. In this recipe, we'll run through the process of installing a plugin, which will prepare us for all the recipes that are to follow.

For our example, we'll install the `contrib-jshint (0.10.0)` plugin. The same steps used to install this plugin can be used to install any of the other plugins available in the plugin package index.

Getting ready

In this example, we'll work with the basic project structure we created in the *Setting up Grunt on a project* recipe of this chapter Be sure to refer to it if you are not yet familiar with it's contents.

How to do it...

The following steps take us through installing a plugin on our project and loading the tasks it contains:

1. The first step of installing a plugin is to install the package that contains it in the current project path. For our example, we'll install the `contrib-jshint` plugin that is contained in the `grunt-contrib-jshint` package. We can install this package and add it to our project dependencies by using the following command:

   ```
   $ npm install --save grunt-contrib-jshint
   ```

2. Next, we'll have to load the tasks contained in the plugin package so they can be used in our configuration. This is done using the `loadNpmTasks` function, provided to us by the `grunt` object that is passed to us in the configuration file. After adding this, our configuration file should look similar to the following:

   ```
   module.exports = function (grunt) {
     grunt.initConfig({});
     grunt.loadNpmTasks('grunt-contrib-jshint');
     grunt.registerTask('default', []);
   };
   ```

3. Now that we have the package installed and its tasks loaded, we can use the loaded tasks in our configuration. For our example, we had the `jshint` task loaded, which enables us to use it in a manner similar to the following:

   ```
   module.exports = function (grunt) {
     grunt.initConfig({
       jshint: {
         sample: {
           files: 'src/*.js'
         }
       }
     });
     grunt.loadNpmTasks('grunt-contrib-jshint');
     grunt.registerTask('default', []);
   };
   ```

There's more...

As you start to make use of more and more Grunt plugins, you will soon start to wonder whether there is a way to optimize the process a little. Fortunately, someone else has already gone down this road before and created the `load-grunt-tasks` utility that automates the loading of tasks from all the packages mentioned in your project dependencies. This means that we no longer need to add a `loadNpmTasks` call for each plugin we install.

The following steps illustrate the usage of this utility, continuing from the work we did earlier in the main recipe:

1. Install the utility's package in the current project path and add it to your dependencies by using the following command:

   ```
   $ npm install --save load-grunt-tasks
   ```

2. Now we can use it in our configuration file by importing the package and passing the `grunt` object to it. Now that we're making use of this utility, we can also remove all the `loadNpmTasks` we were using to load our plugins. This should result in a configuration file similar to the following:

   ```
   module.exports = function (grunt) {
     require('load-grunt-tasks')(grunt);
     grunt.initConfig({
       jshint: {
         sample: {
           files: 'src/*.js'
         }
       }
     });
     grunt.registerTask('default', []);
   };
   ```

   ### Tip

   The `load-grunt-tasks` plugin will, by default, only load plugins that have names with `grunt` at their start. This behavior can be customized by using the `pattern` option. To find out more about the `load-grunt-tasks` plugin, refer to the plugin's page at the following URL:

   https://github.com/sindresorhus/load-grunt-tasks

Setting up a basic web server

A simple web server will always come in handy during the development process of web-based projects. They can be easily set up and used to serve web content from your local machine, so you don't have to worry about constantly deploying your experimental changes to a remote service provider.

We'll make use of the `contrib-connect (0.8.0)` plugin, which provides us with the ability to set up and run a simple web server based on the **Connect** server framework. By default, it will only serve files from a directory, but it has the added benefit of being able to make use of the many Connect middleware plugins available.

Tip

You can read more about the Connect server framework and its middleware plugins at the following URL:

https://github.com/senchalabs/connect

Getting ready

In this example, we'll work with the basic project structure we created in the *Setting up Grunt on a project* recipe of this chapter. Be sure to refer to it if you are not yet familiar with it's contents.

How to do it...

The following steps take us through setting up a development server that serves files from a directory located in our project directory.

1. We'll start by installing the package containing the `contrib-connect` plugin and loading its tasks as given in the instructions provided in the *Installing a plugin* recipe of this chapter.
2. With the `connect` task loaded from the plugin, we can make use of it in our configuration. To do this, we add the following to our configuration:

   ```
   connect: {
     server: {
       options: {
         base: 'www',
         keepalive: true
       }
     }
   }
   ```

   ### Tip

   The `base` option indicates in which directory the server should look for the files that are requested. Everything that you'd like to serve with your development server can be placed in this directory. This includes HTML pages, JavaScript source files, style sheets, images, and so on.

   The `keepalive` option keeps the server running even if the requested task has finished. This is usually preferred if you're running the `connect` task on its own, but is not required if another task will be running indefinitely after it has completed.

3. Let's add a simple file that we'd like to have served from our server so that we have something to test it with. Create a directory called `www` in the project root, and then a file inside it called `index.html`, with the following contents:

   ```
   <html>
     <head>
       <title>Test Page</title>
     </head>
     <body>
       <h1>This is a test page.</h1>
     </body>
   </html>
   ```

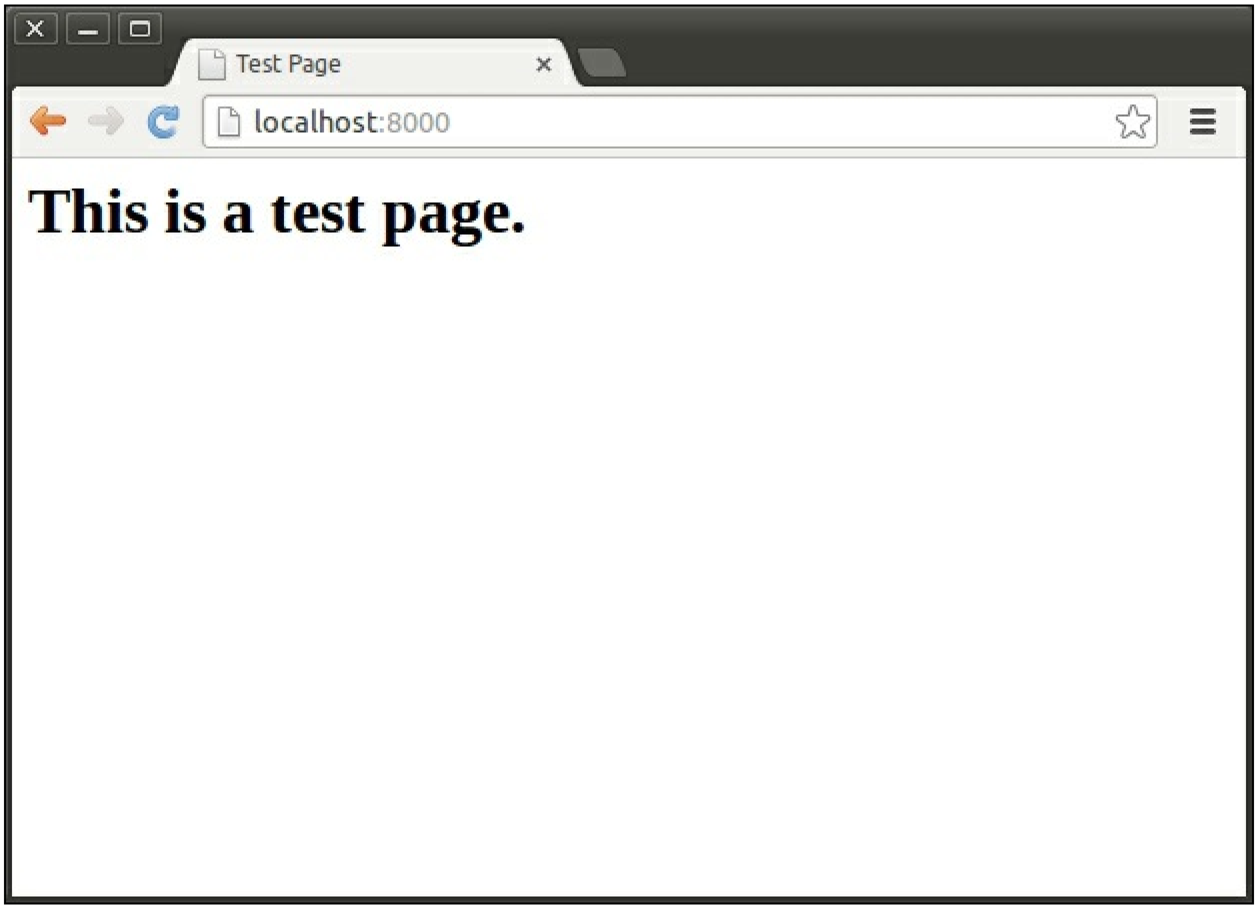
Tip

Like many other web servers, the Connect server started by this task will always look for an `index.html` file in a folder if no filename is specified in the request URL.

4. Now we can run our web server using the `grunt connect` command, which will produce output indicating that our server started, along with a URL of where it can be reached:

```
Running "connect:server" (connect) task
Waiting forever...
Started connect web server on http://0.0.0.0:8000
```

5. Finally, we can use our favorite browser to pay a visit to the URL mentioned in the output. This would show us our example page, as served through our running server:



This is a test page.

There's more...

The `connect` task provides many useful configuration options that allow us to automatically open a browser, specify the port and hostname where the server should run, serve files from more than one directory, use other Connect middleware plugins, and adding extra functionality to the created server.

Opening the default web browser on the default URL

In order to automatically open our favorite web browser on the default URL that our web server provides, we can set the `open` option to `true` as we do in the following example:

```
options: {
  base: 'www',
  keepalive: true,
  open: true
}
```

Opening a specific web browser at a specific URL

When we'd like to use a different web browser from the default one to open a URL other than the default one, we can provide an object to the `open` option that specifies exactly what we'd like. The following code specifies a URL, the browser that should be used to open it, and a callback function that should be called once it's been opened:

```
options: {
  base: 'www',
  keepalive: true,
  open: {
    target: 'http://localhost:8000/test.html',
    appName: 'firefox',
    callback: function() {
      console.log('Test URL opened in Firefox!');
    }
  }
}
```

Tip

Downloading the example code

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

Using a specific port

The default port used by the connect task is `8000`, but if you'd like to use a different one, the `port` option can be used to specify which one you'd like it to be, which is shown in the following code snippet:

```
options: {
  port: 3000,
  base: 'www',
  keepalive: true
}
```

Automatically selecting an available port

In case you are not sure about which ports will be available when you start the server, it can be quite useful for you to have the server automatically select an open port. Setting the `useAvailablePort` option to `true` will enable this behavior. The code snippet for this is as follows:

```
options: {
  base: 'www',
  keepalive: true,
  useAvailablePort: true
}
```

Using a specific hostname

In the case that you'd like the server to attach to a specific hostname and not just the default `0.0.0.0`, you can make use of the `hostname` option in the following manner:

```
options: {
  base: 'www',
  keepalive: true,
  hostname: 'something.else.com'
}
```

Serving files from multiple directories

If you have more than one directory that contains the files that you'd like to be serving, then the `base` option can be provided with an array of directory names to look up content. Here is a code snippet for your reference:

```
options: {
  base: ['www', 'other'],
  keepalive: true
}
```

Tip

When using an array for the `base` option, the server will look up the requested resources in each of the directories, from left to right, and return a resource as soon as it's found. If each of the two directories in the example contained an `index.html` file, a request to the root URL would return the `index.html` file in the `www` directory.

Using middleware

If we'd like to use one of the many existing middleware plugins available for the Connect framework, we can set the `middleware` option to a function that modifies the middleware stack by adding the desired middleware to it.

1. First, we'll need to install the middleware that we'd like to use, which in our case is packaged along with the Connect server framework. We can install the framework package using the following command:

   ```
   $ npm install --save connect
   ```

2. Now, we alter the options of the `server` target in the `connect` task so that it adds the `compress` middleware to the stack:

   ```
   options: {
     base: 'www',
     keepalive: true,
     middleware: function(connect, options, middlewares) {
       middlewares.push(
         require('connect').middleware.compress()
       );
       return middlewares;
     }
   }
   ```

   ### Tip

   The `middleware` option can also be set to an array, but this will replace the default stack that is provided by the `connect` task. The default middleware allows the serving of files from the directories indicated by the `base` option.

Adding functionality to the created server

There are times that it would be useful to work with the server that is created by the `connect` task. A good example of this is when we'd like to enable our server to handle **Socket.IO** interactions. This can be done by providing a function to the `onCreateServer` option that works with the created server in whichever way you like:

```
options: {  base: 'www',
  keepalive: true,
  onCreateServer: function(server, connect, options) {
    var io = require('socket.io').listen(server);
    io.sockets.on('connect', function (socket) {
      // do something with socket
    });
  }
}
```

Tip

This example assumes that you've already installed the `socket.io` package. You can find out more about `socket.io` at its website:

http://socket.io/

Watching files for changes

Another common requirement for development environments is the need to automatically run specific tasks when certain files are changed. This is especially useful when you'd like to monitor the quality of changing code in real time, or recompile altered resources as soon as they change, so that the effect of the changes are reflected without any manual intervention.

The `contirb-watch (0.6.1)` plugin allows us to keep a watch on a specific set of files, and run a specified set of tasks when file events are observed.

Getting ready

In this example, we'll work with the basic project structure that we created in the *Setting up Grunt on a project* recipe of this chapter. Be sure to refer to it if you are not yet familiar with it's contents.

How to do it...

The following steps take us through setting up a `watch` task that initiates a code quality analysis on a JavaScript source file each time a change to the file is observed.

1. We'll start by installing the package that contains the `contrib-watch` plugin and loading its tasks by following the instructions provided in the *Installing a Plugin* recipe of this chapter.
2. For our example, we'll make use of the `jshint` task to analyze the quality of a JavaScript source file. Let's install the `contrib-jshint` plugin and load its tasks by following the instructions provided in the *Installing a Plugin* recipe of this chapter.
3. We'll also need a JavaScript source file that we can watch for changes and perform a quality analysis on. Let's create a file called `sample.js` in our project root and provide it with the following contents:

   ```
   var sample = 'Sample';
   console.log(sample);
   ```

4. Now, we can set up the example `jshint` task, which we'll run using the `watch` task by adding the following to our configuration:

   ```
   jshint: {
     sample: {
       src: ['sample.js']
     }
   }
   ```

5. With the plugin installed and the sample task configured, we can now configure a target on the `watch` task, which will run the `jshint` task every time the sample file called `sample.js` changes. This is done by adding the following to our configuration:

   ```
   watch: {
     sample: {
       files: ['sample.js'],
       tasks: ['jshint']
     }
   }
   ```

6. Finally, we can start the task using the `grunt watch` command, which should produce the following output to confirm that it's running:

   ```
   Running "watch" task
   Waiting...
   ```

7. To test our setup, we can now make some changes to the `sample.js` file and save them. This should produce output informing us of the file event similar to the following:

   ```
   Running "watch" task
   Waiting...
   >> File "sample.js" changed.
   Running "jshint:sample" (jshint) task
   >> 1 file lint free.

   Done, without errors.
   Completed in 1.0s at Wed Jan 1 2014 00:00:00 GMT - Waiting...
   ```

There's more...

The `watch` task plugin provides many more useful configuration options that allow us to watch more than one file, run a series of tasks, prevent process spawning for task runs, enable the interruption of task runs, specify the waiting period before rerunning tasks, run tasks only on specific file events, allow tasks to kill the watcher process, and run tasks once when the watcher starts up.

Watching more than one file

In case we'd like to watch more than one file, the pattern-matching capabilities of the standard Grunt `files` configuration can be used. The following configuration example will observe all the files in the project root or any of its sub directories with the `txt` extension:

```
watch: {
  sample: {
    files: ['**/*.txt'],
    tasks: ['sample']
  }
}
```

Tip

You can find out more about the file's configuration and the globbing patterns it supports at the following URL:

http://gruntjs.com/configuring-tasks#files

Running a series of tasks

In case we'd like to run more than one task each time a file event is observed, we can just add the tasks to the array passed to the `tasks` configuration:

```
watch: {
  sample: {
    files: ['sample.txt'],
    tasks: ['sample', 'another', 'finally']
  }
}
```

Tip

The tasks specified in using the `tasks` configuration are run one at a time, in the order they are placed inside the array.

Preventing process spawning for task runs

The default behavior of the `watch` task is to start each of the triggered tasks in their own child process. This prevents failing triggered tasks from causing the `watch` task itself to fail. As a side effect, it also clones the context of the watcher process for each task. This behavior can however be disabled by setting the `spawn` option to `false`, which triggers tasks a little faster and allows them to share a context between them. The following demonstrates the configuration for this:

```
watch: {
  sample: {
    files: ['sample.txt'],
    tasks: ['sample'],
    options: {
      spawn: false
    }
  }
}
```

Enabling the interruption of task runs

The default behavior for the watcher is to wait for the completion of the tasks triggered by the previous change, before waiting for changes again. By setting the `interrupt` option to `true`, the watcher will stop running tasks when a change is detected and start rerunning them. The following demonstrates the configuration for this:

```
watch: {
  sample: {
    files: ['sample.txt'],
    tasks: ['sample'],
    options: {
      interrupt: true
    }
  }
}
```

Specifying the waiting period before rerunning tasks

The default period the watcher will wait before checking for file changes after a previous task run is `500ms`. This amount of time can be changed by setting the `debounceDelay` option to the number of milliseconds you'd like for it to wait. The following demonstrates the configuration for this:

```
watch: {
  sample: {
    files: ['sample.txt'],
    tasks: ['sample'],
    options: {
      debounceDelay: 1000
    }
  }
}
```

Run tasks only on specific file events

In addition to being changed, files can also be added and deleted. The default behavior of the watcher is to observe all these events, but if you'd like it to run tasks only on specific events, the `event` option can be set to either `changed`, `added`, `deleted`, or `all`.

The following example will only start the `sample` task if a file named `sample.txt` is added or deleted to the same path as the configuration file:

```
watch: {
  sample: {
    files: ['sample.txt'],
    tasks: ['sample'],
    options: {
      event: ['added', 'deleted']
    }
  }
}
```

Allowing tasks to kill the watcher process

Warnings and failures that are raised by the tasks started by the watch task will, by default, not interrupt its execution. Setting the `forever` option to `false` will disable this behavior and allow child tasks to stop the watcher process on warnings and failures. The following demonstrates the configuration for this:

```
watch: {
  options: {
    forever: false
  },
  sample: {
    files: ['sample.txt'],
    tasks: ['sample']
  }
}
```

Tip

Note that the `forever` option is a task-level option only and cannot be specified for individual targets.

Running tasks once at startup of the watcher

In the case that you'd like to run the tasks specified in the `tasks` option once the watcher starts up, and not only once file events are observed, you can set the `atBegin` option to `true`. The following demonstrates the configuration for this:

```
watch: {
  sample: {
    files: ['sample.txt'],
    tasks: ['sample'],
    options: {
      atBegin: true
    }
  }
}
```

Setting up LiveReload

Once you've got a development server running that serves the pages, code, and assets that make up your web application, you will notice that you constantly have to refresh the browser each time you wish to observe a change that was made.

This is where the **LiveReload** tool and its constituent libraries come in handy. It's designed to automatically reload the browser's contents if a page or code file is changed, and even apply changes to CSS and images live, without refreshing the browser's contents.

We can set up LiveReload for our project with the help of two plugins that are discussed in other parts of this chapter. The development server provided by the `contrib-connect (0.8.0)` plugin can be configured to accept LiveReload triggers, and the `contrib-watch (0.6.1)` plugin to send them file events.

Getting ready

In this example, we'll work with the basic project structure that we created in the *Setting up Grunt on a project* recipe of this chapter. Be sure to refer to it if you are not yet familiar with it's contents.

We also make use of the `contrib-connect` and `contrib-watch` plugins in this recipe, which have each been discussed separately in the *Setting up a basic web server* and *Watching files for changes* recipes of this chapter. These are to be referred to if you are not yet familiar with the plugins they discuss.

How to do it...

The following steps take us through setting up a watch task that will automatically trigger a refresh when it observes changes to an HTML page that is served from a local development server.

1. We'll start by installing the packages containing the `contrib-connect` and `contrib-watch` plugins and loading their tasks by following the instructions provided in the *Installing a plugin* recipe of this chapter.
2. For the sake of our example, we'll create a file called `index.html` in the `www` directory, which will be the file that we wish to view in a browser and have automatically updated when changes are made to it. The contents for this file follows:

   ```
   <html>
     <head>
       <title>LiveReload Test</title>
     </head>
     <body>
       <h1>First there was this.</h1>
     </body>
   </html>
   ```

3. Next, we'll set up our development server, which will serve the `index.html` file from the `www` directory. Along with the standard configuration, we'll set the `livereload` option to `true`, indicating that the development server should enable the browser to receive LiveReload triggers. This is all done by adding the following to our configuration:

   ```
   connect: {
     dev: {
       options: {
         base: 'www',
         livereload: true
       }
     }
   }
   ```

   ### Tip

   Providing the `true` value for the `livereload` option includes `connect-livereload` in the middleware stack of the connect server. The middleware in turn inserts a snippet of code in the HTML code of the pages served, which enables the browser to accept LiveReload triggers.

The `keepalive` option can be excluded from this configuration due to the watcher process that will continue to run after the server is started. This means that the Grunt process will not end, which would also have stopped the server it started.

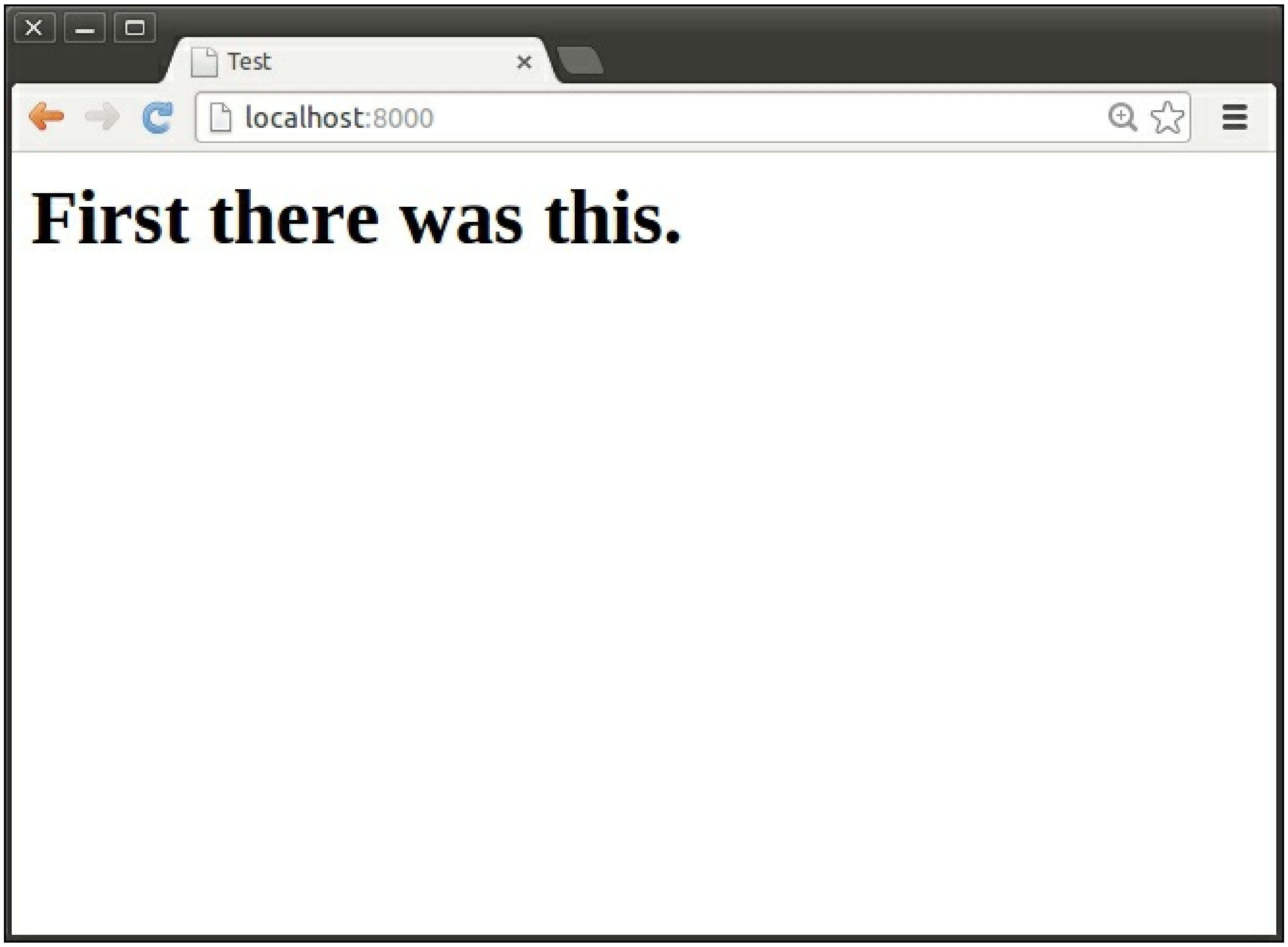
4. Now, we'll set up a watcher to observe file events in the `www/index.html` file. Along with the standard configuration, we'll set the `livereload` option to `true`, indicating that the appropriate LiveReload triggers should be sent whenever changes are observed. This is done by adding the following to our configuration:

```
watch: {
  www: {
    files: ['www/index.html'],
    options: {
      livereload: true
    }
  }
}
```

5. Finally, we can start our server and watcher using the `grunt connect watch` command, which should produce output indicating the start of both:

```
Running "connect:dev" (connect) task
Started connect web server on http://0.0.0.0:8000

Running "watch" task
Waiting...
```

6. Now, we can use our favorite browser to open the URL mentioned in the output, which should show us our sample page as served by the server:

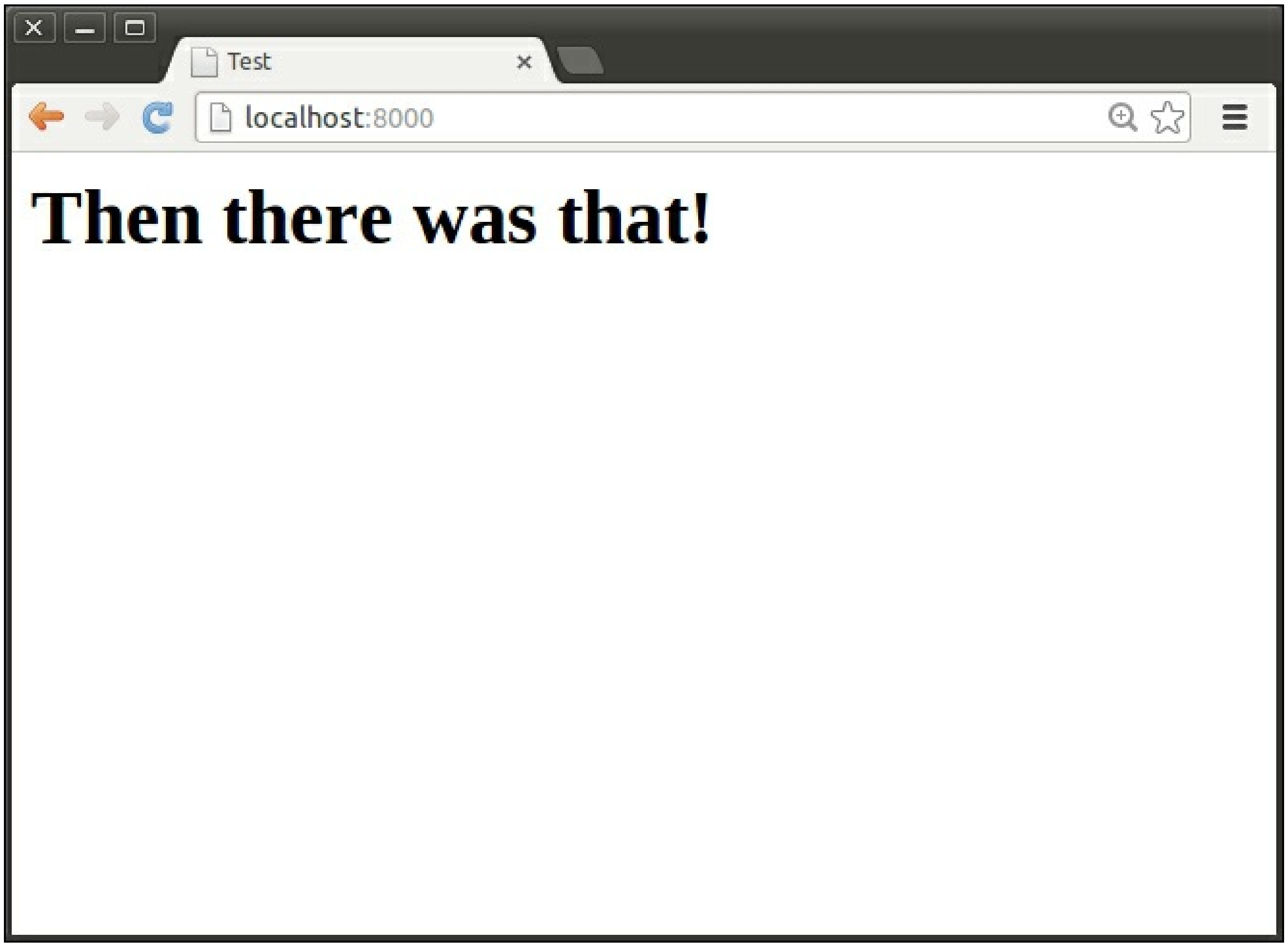


First there was this.

7. Let's try out the LiveReload functionality by changing the `www/index.html` file and saving it. This action should produce the following output in the terminal in which the server and watcher were started:

```
>> File "www/index.html" changed.
Completed in 0.001s at Wed Jan 01 2014 00:00:00 GMT - Waiting...
```

8. Switching back to our browser that currently has the `http://0.0.0.0:8000` URL open, we should now see the updated page, without having initiated a manual refresh:



Then there was that!

Processing only changed files

When running a task that processes files in one way or another, you'll soon realize that you probably don't want it to process all the files each time it is performed. This is especially true when the amount of files the task has to process becomes quite large.

This is where the `newer (0.7.0)` plugin can help out by ensuring that only the files that have changed since the task's previous run are processed, each time it is called to run. It can be used with any plugin that makes use of the standard `files` configuration and becomes especially useful when using a watcher to rerun tasks each time a file change is detected.

Getting ready

In this example, we'll work with the basic project structure that we created in the *Setting up Grunt on a project* recipe of this chapter. Be sure to refer to it if you are not yet familiar with it's contents.

How to do it...

The following steps take us through making use of the `newer` plugin to check the code quality of only the JavaScript source files that have changed since the last run.

1. We'll start by installing the package containing the `newer` plugin and loading its tasks by following the instructions provided in the *Installing a plugin* recipe of this chapter.
2. For the purpose of our example, we'll also install the `contrib-jshint` plugin and load its tasks by following the instructions provided in the *Installing a plugin* recipe of this chapter.
3. With all the required plugins installed, we can now add sample configuration for the `jshint` task, which will perform a code quality check on all the JavaScript files in the `src` directory. This is done by adding the following to our configuration:

   ```
   jshint: {
     sample: {
       src: 'src/**/*.js'
     }
   }
   ```

   Tip

   Note that the `jshint` plugin requires you to specify the files that are to be targeted by the task in the `src` configuration directly inside the target. In other cases, it is usually recommended to specify your target files using the `files` configuration.

4. Now we can run the `jshint` task a couple of times using the `grunt jshint` command to observe its behavior. Each time we run it, we'll see that it scans all the files in the `src` directory. This should produce the same output each time, similar to the following:

   ```
   Running "jshint:sample" (jshint) task
   >> 3 files lint free.
   ```

5. In order to make use of the `newer` plugin, we prepend `newer:` to the name of the task we wish to run. On running the `grunt newer:jshint` command the first time, the `newer` plugin will cache the timestamps of the files that have been processed. This produces output similar to the following:

   ```
   Running "newer:jshint" (newer) task
   ```

```
Running "newer:jshint:sample" (newer) task

Running "jshint:sample" (jshint) task
>> 3 files lint free.

Running "newer-postrun:jshint:sample:.cache" (newer-postrun) task
```

6. When we run the `grunt newer:jshint` command again, we'll see that no files are processed by the `jshint` task, which produces output similar to the following:

   ```
   Running "newer:jshint" (newer) task

   Running "newer:jshint:sample" (newer) task
   No newer files to process.
   ```

7. Now, we can change one of the files in the `src` directory and run the command again to see that the changed file is processed again:

   ```
   Running "newer:jshint" (newer) task

   Running "newer:jshint:sample" (newer) task

   Running "jshint:sample" (jshint) task
   >> 1 file lint free.

   Running "newer-postrun:jshint:sample:1:.cache" (newer-postrun)
   task
   ```

There's more...

Processing only changed files becomes especially useful when making use of the `watch` task provided by the `contrib-watch` plugin. A `watch` task will rerun its indicated tasks every time it observes a file change, which can happen quite often during development, and can take quite a bit of time if the tasks target a large number of files.

The following steps provide an example of how to use the `newer` plugin in conjunction with `contrib-watch` and continues with the work we did in the main recipe:

1. We'll start by installing the package that contains the `contrib-watch` plugin and loading its tasks by following the instructions provided in the *Installing a plugin* recipe of this chapter.
2. Now, we'll add a `watch` task, which will run the `jshint` task when it observes changes on any of the JavaScript files contained in the `src` directory. We'll also prepend the `jshint` task with `newer:` to indicate that we only want to process the files that actually changed. This is done by adding the following to our configuration:

   ```
   watch: {
     jshint: {
       files: ['src/**/*.js'],
       tasks: ['newer:jshint']
     }
   }
   ```

3. Now, we can start the `watch` task by using the `grunt watch` command in the terminal. This should produce output similar to the following, indicating that the watcher is running:

   ```
   Running "watch" task
   Waiting...
   ```

4. If we now change and save a file in the `src` directory, we should see output similar to the following, indicating that only the file that was changed has been processed by the `jshint` task:

   ```
   >> File "src/file.js" changed.
   Running "newer:jshint" (newer) task

   Running "newer:jshint:sample" (newer) task

   Running "jshint:sample" (jshint) task
   >> 1 file lint free.
   ```

```
Running "newer-postrun:jshint:sample:1:.cache" (newer-postrun)
task
```

Importing external data

In most coding practices, it's best practice to keep logic and data separated as much as possible. The same rule applies to the Grunt configuration logic and the data that it makes use of.

A very common use case for using data from an external source is the use of the project information contained in the `package.json` file. Information such as project names and version numbers might not change too often, but when they do, we'd probably prefer not having to look for them everywhere in the project.

Fortunately, the Grunt framework provides us with functions that allow us to easily read data from external files and store them in the configuration. This stored data can then also be easily used with the help of the string templates that are automatically processed for all configurations.

Getting ready

In this example, we'll work with the basic project structure that we created in the *Setting up Grunt on a project* recipe of this chapter. Be sure to refer to it if you are not yet familiar with it's contents.

How to do it...

The following steps take us through making use of data from the `package.json` file when generating an optimized version of a JavaScript source file.

1. For our example, we'll make use of the `contib-uglify` plugin, which can be used to compress JavaScript source files. Let's install it and load its tasks for using the instructions provided in the *Installing a Plugin* recipe of this chapter.
2. We'll also need a simple JavaScript source file for the sake of our example. Let's create a file called `sample.js` in the root of our project directory and fill it with the following code:

   ```
   module.exports = function () {
     var sample = 'Sample';
     console.log(sample);
   };
   ```

3. Next, we'll import the data contained in our project's `pacakge.json` file by making use of the `grunt.file.readJSON` function and assigning its result to a property called `pkg` in our configuration. After adding this, our configuration object should look similar to the following:

   ```
   {
     pkg: grunt.file.readJSON('package.json')
   }
   ```

   ### Tip

   Note that the property name `pkg` is just used for this example, and can be pretty much anything except for the names of the tasks that are available for configuration.

4. Now that we have the data imported form our project package, we can set up an `uglify` task, which will compress a JavaScript source file called `sample.js`, using the version number contained in the `package.json` file as part of the resulting file's name. This is done by adding the following to our configuration:

   ```
   uglify: {
     sample: {
       files: {
         'sample_<%= pkg.version %>.js': 'sample.js'
       }
     }
   }
   ```

   ### Tip

Grunt makes use of the **Lo-Dash** string template system. You can read more about it at the following URL:

http://lodash.com/docs/#template

5. Finally, we can use the `grunt uglify` command to test our setup, which should produce output similar to the following:

```
Running "uglify:sample" (uglify) task
File sample_0.0.0.js created: 81 B → 57 B
```

6. We can now also take a look at our newly generated compressed version of the `sample.js` file in the `sample_0.0.0.js` file, which should have content similar to the following:

```
module.exports = function(){var a="Sample";console.log(a)};
```

There's more...

The YAML format provides another popular way to store human readable data and can also be easily imported into our configuration. The following example, based on the work we did in the main recipe, demonstrates this functionality:

1. First, we'll create a simple YAML file for the purpose of our example. Let's create `sample.yaml` in our project root and give it the following content:

   ```
   version: 0.0.0
   ```

2. Now all we need to do is change the call to `grunt.file.readJSON` to import our sample YAML file instead. We do this by changing the `pkg` configuration to the following:

   ```
   pkg: grunt.file.readYAML('sample.yaml')
   ```

3. If we now run the Grunt `uglify` command, we should see the same result as before, with output similar to the following:

   ```
   Running "uglify:sample" (uglify) task
   File sample_0.0.0.js created: 81 B → 57 B
   ```

Chapter 2. File Management

In this chapter, we will cover the following recipes:

- Copying files
- Compressing files
- Creating symbolic links
- Concatenating files
- Fetching a single URL
- Fetching multiple URLs

Introduction

In this chapter, we'll be focused on making use of Grunt for our every-day file management tasks. A day rarely passes without us copying, compressing, downloading, or concatenating files, and with the help of Grunt, we can automate even these simple tasks.

Copying files

In our quest for ultimate project automation, it won't be long before we'll want to automate the copying of files or directories from one part of our project to another. The `contrib-copy(0.5.0)` plugin provides us with this functionality, along with some other options that become especially useful in the process of copying files.

Getting ready

In this example, we'll work with the basic project structure we created in the *Installing Grunt on a project* recipe in Chapter 1, *Getting Started with Grunt*. Be sure to refer to it if you are not yet familiar with its contents.

How to do it...

The following steps will take us through copying a single file from one directory to another.

1. We'll start by installing the package that contains the `contrib-copy` plugin as per the instructions provided in the *Installing a plugin* recipe in Chapter 1, *Getting Started with Grunt*.
2. Let's also set up a sample file in our project, which we can try out with the copy operation. Create a file named `sample.txt` in the `src` directory with the following contents:

   ```
   Some sample data.
   ```

3. Now, we can add the configuration for our `copy` task, which will instruct it to copy our sample file from the `src` directory to the `dest` directory:

   ```
   copy: {
     sample: {
       src: 'src/sample.txt',
       dest: 'dest/sample.txt'
     }
   }
   ```

   ### Tip

   Note that the destination directory specified in the copy operation will be automatically created if it does not yet exist.

4. Finally, we can run the `grunt copy` command, which should provide output informing us of the successful copy operation:

   ```
   Running "copy:files" (copy) task
   Copied 1 files
   ```

5. To confirm that the sample file was copied, we can now open the `dest/sample.txt` file using our favorite editor and confirm that its contents match the original file's contents.

There's more...

The `copy` task provides some useful configuration options that allow us to process the contents of the files that are being copied, copy the source file's permissions, and set the file permissions for destination files.

Processing file contents

The `copy` task provides the option of altering the contents of files as they are being copied. There are a number of useful applications for this functionality, such as rendering a template or replacing parts of a file that are matched by a regular expression.

In the following example, we'll simply focus on prepending a string to a copied file. We'll build upon our work earlier in this recipe.

1. We'll start by providing a stub function for the `process` option that just passes through the content without altering anything. This will change the configuration of our `copy` task to the following:

```
copy: {
  sample: {
    options: {
      process: function (content, path) {
        return content;
      }
    },
    src: 'src/sample.txt',
    dest: 'dest/sample.txt'
  }
}
```

2. Now, we can alter the function passed to the `process` option so that it prepends a string to the contents of the file being copied, making it look similar to the following:

```
process: function (content, path) {
  return 'Prepended string!\n' + content;
}
```

3. If we now run the `grunt copy` command again, we will see a confirmation of the file being copied, and looking at the contents of the `dest/sample.txt` file, we should see the following:

```
Prepended content
```

```
    Sample content.
```

Copying the source file's permissions

The permissions attached to a file are usually of great importance. By default, the copy task will ignore the file permissions of the source file, but if you'd like them to be applied to the destination file, then the `mode` option can be set to `true` as per the following example:

```
copy: {
  sample: {
    options: {
      mode: true
    },
    src: 'src/sample.txt',
    dest: 'dest/sample.txt'
  }
}
```

Setting the permissions of the destination file

In case you'd like to specify the permissions that the destination files of a copy operation should get, you can do so by setting the `mode` option to the desired permission mode using the traditional Unix permissions in octal format.

Tip

You can read more about the traditional Unix permissions convention at http://en.wikipedia.org/wiki/File_system_permissions.

There's also a handy Unix file permissions calculator available at http://permissions-calculator.org/.

The following example configuration will make the destination file readable and writable, only for their owner:

```
copy: {
  files: {
    src: 'src/sample.txt',
    dest: 'www/sample.txt',
    options: {
      mode: '0600'
    }
  }
}
```

Compressing files

The compression of files is quite a common practice within our local system, and across the World Wide Web. The most prominent use for file compression is to reduce the size of data that will be transferred across a network.

For our own purposes, we may at some point want to automate the compression of a file or folder that we'd like to make available for download or prepare for a transfer. The `contrib-compress (0.10.0)` plugin provides us with this functionality, along with options to specify the archive type and algorithms used in the process.

Getting ready

In this example, we'll work with the basic project structure we created in the *Installing Grunt on a project* recipe in Chapter 1, *Getting Started with Grunt*. Be sure to refer to it if you are not yet familiar with its contents.

How to do it...

The following steps will take us through compressing a single sample file using the default settings of the `compress` task.

1. We'll start by installing the package that contains the `contrib-compress` plugin as per the instructions provided in the *Installing a plugin* recipe in Chapter 1, *Getting Started with Grunt*.
2. Then, we can create a sample file to demonstrate the compression functionality. Let's download a book in text format as an example and save it as `book.txt` in our project directory.

   ### Tip

   The text version of Jane Austen's *Pride and Prejudice* can be downloaded at the following URL:

   http://www.gutenberg.org/cache/epub/1342/pg1342.txt
3. Now, we can add the following configuration for the `compress` task, which will compress our sample file into the `book.txt.gz` archive file:

   ```
   compress: {
     sample: {
       src: 'book.txt.gz',
       dest: 'book.txt'
     }
   }
   ```
4. To try it out, we can run the `grunt compress` command, which should produce output similar to the following:

   ```
   Running "compress:sample" (compress) task
   Created book.txt.gz (314018 bytes)
   ```
5. If we take a look at the `book.txt.gz` file, we'll see that it's about half the size of the `book.txt` file. We can also go as far as extracting this file using our favorite archive extractor to confirm that it contains the original file.

There's more...

The `compress` task provides us with several options that allow us to archive a collection of files, specify the compression mode, and specify the archive compression level.

Archiving a collection of files

In our main recipe, we compressed a single file on its own, but it's possibly more common to archive a collection of files into one single file. In order to do this, we alter our `src` configuration to specify the files that should be archived, remove the `dest` configuration, and then provide a filename for our archive file using the `archive` option.

The following steps continues from the work we did earlier in this recipe, through to archiving and compressing a collection of files into a single file.

1. First, we'll add some more files that we'd like to have archived and compressed. Let's change the name of our previously downloaded book from `book.txt` to `austen.txt` and download a couple more, renaming them as `wells.txt` and `thoreau.txt`.

   #### Tip

   The two sample books mentioned can be downloaded from the following URLs in the same order they were mentioned:

   http://www.gutenberg.org/cache/epub/36/pg36.txt

   http://www.gutenberg.org/files/205/205-0.txt

2. Now, we can change our `compress` task's configuration from just compressing the file named `book.txt` to including all three downloaded books in an archive file named `books.zip`. We do this by changing its configuration to the following:

```
compress: {
  sample: {
    options: {
      archive: 'books.zip'
    },
    src: ['austen.txt', 'wells.txt', 'thoreau.txt']
  }
}
```

3. Let's try it out by running the `grunt compress` command, which should provide us with output similar to the following:

   ```
   Running "compress:sample" (compress) task
   Created books.zip (786932 bytes)
   ```

4. We should now see the `books.zip` archive in our project directory. To confirm that it contains all the files we specified, you can use your favorite archiving tool to explore or extract it.

Specifying the compression mode

By default, the `compress` task will determine the compression mode to use in its operations by looking at the extension of the file name specified in `archive` option. This behavior can be altered by indicating the compression mode using the `mode` option.

The following example will make use of the `zip` mode despite the extension of the filename specified in the `archive` option being `tgz`:

```
compress: {
  sample: {
    options: {
      mode: 'zip',
      archive: 'books.tgz'
    },
    files: {
      src: ['austen.txt', 'wells.txt', 'thoreau.txt']
    }
  }
}
```

Tip

At the time of publication, the compression modes supported for the `mode` option were `gzip`, `deflate`, `deflateRaw`, `tar`, `tgz`, and `zip`.

Specifying the archive compression level

In case we'd like to specify a compression rate when using either the `zip` or `gzip` compression modes, we can do so using the `level` option. This option is set to `1` by default, but can be set to any integer up to `9`, the latter indicating better compression at the cost of slower performance.

The following configuration will compress our books to level `9`, which takes a little longer, but decreases the size of the produced archive by about 10 percent:

```
compress: {
  sample: {
    options: {
      level: 9,
      archive: 'books.zip'
    },
    src: ['austen.txt', 'wells.txt', 'thoreau.txt']
  }
}
```

Creating symbolic links

Creating references to files and directories can be quite useful in various circumstances, most notably when we'd like to distribute duplicates of a file and keep them up to date without having to manually copy them after each change.

The functionality of creating symbolic links is provided to us by the `contrib-symlink (0.3.0)` plugin. It's a very simple plugin but provides all the standard Grunt configuration options.

Getting ready

In this example, we'll work with the basic project structure we created in the *Installing Grunt on a project* recipe in Chapter 1, *Getting Started with Grunt*. Be sure to refer to it if you are not yet familiar with its contents.

How to do it...

The following steps will take us through creating a symbolic link to a file named `assets/img/logo.png` inside the `www/img` directory.

1. We'll start by installing the package that contains the `contrib-symlink` plugin as per the instructions provided in the *Installing a plugin* recipe in Chapter 1, *Getting Started with Grunt*.
2. Then, we can add the following configuration, which will create the symbolic link in the `img/www` directory:

   ```
   symlink: {
     sample: {
       files: {
         'www/img/logo.png': 'assets/img/logo.png'
       }
     }
   }
   ```

3. To try it out, we can run the `grunt symlink` command, which should inform us of the creation of the symbolic link with output similar to the following:

   ```
   Running "symlink:files" (symlink) task
   >> Created 1 symbolic links.
   ```

4. If you now take a look inside the `www/img` directory, you should see a symbolic link called `logo.png`, which points to the `assets/img/logo.png` file. You can perhaps also try to change the contents of the image to see them reflected in the symbolic link.

Concatenating files

The practice of concatenating files is quite often found in the routines of all kinds of developers. Be it combining fragmented log files or many smaller source files into one big code base, joining files end-to-end comes in handy on a regular basis.

Concatenation functionality can be provided to us by the `contrib-concat (0.4.0)` plugin. Along with the standard Grunt configurations, it also provides a large set of options to tailor its behavior according to our unique requirements.

Getting ready

In this example, we'll work with the basic project structure we created in the *Installing Grunt on a project* recipe in Chapter 1, *Getting Started with Grunt*. Be sure to refer to it if you are not yet familiar with its contents.

How to do it...

The following steps will take us through concatenating three JavaScript source files into one combined source file, so that we only have to serve one source file along with our web application.

1. We'll start by installing the package that contains the `contrib-concat` plugin as per the instructions provided in the *Installing a plugin* recipe in Chapter 1, *Getting Started with Grunt*.
2. For our example, we'll also require three JavaScript source files. Let's call them `one.js`, `two.js`, and `three.js`, and provide them each with the following contents, only replacing `<filename>` with their names:

   ```
   var sample = 'Sample code from <filename>.';
   console.log(sample);
   ```

3. Now, we can add the following configuration for the `concat` task, which will combine our three sample files into the `all.js` file:

   ```
   concat: {
     sample: {
       files: {
         'all.js': ['one.js', 'two.js', 'three.js']
       }
     }
   }
   ```

4. To try out our task, we can run the `grunt concat` command, which will produce output similar to the following:

   ```
   Running "concat:sample" (concat) task
   File all.js created
   ```

5. We can now take a look at the `all.js` file that was created, which will contain the following code:

   ```
   var sample = 'Sample code from one.js.';
   console.log(sample);
   var sample = 'Sample code from two.js.';
   console.log(sample);
   var sample = 'Sample code from three.js.';
   console.log(sample);
   ```

There's more...

The `concat` task provides us with several options that allow us to join the files with a custom separator, strip banners before concatenation, add a banner and footer to the concatenated result, and process the source file's contents before concatenation.

Joining files with a custom separator

It's usually preferable to have a prominent separator between the joined files so that it's easier to distinguish between them, in case we need to review the contents of the concatenated result.

By default, the `concat` task will just separate joined files with a linefeed character, but a custom separator can be provided using the `separator` option as per the following example:

```
concat: {
  sample: {
    options: {
      separator: '\n\n/* Next file */\n\n'
    },
    files: {
      'all.js': ['one.js', 'two.js', 'three.js']
    }
  }
}
```

Running the task with the preceding example code should produce a `all.js` file with contents similar to the following:

```
var sample = 'Sample code from one.js.\n';
console.log(sample);

/* Next file */

var sample = 'Sample code from two.js.\n';
console.log(sample);

/* Next file */

var sample = 'Sample code from three.js.\n';
console.log(sample);
```

Stripping banners before concatenation

The need to remove banners from JavaScript source files before joining them is quite common since they usually take up quite a bit of extra storage space, and as every web developer knows, every bit counts.

To automatically strip banners from JavaScript source files, we can make use of the `stripBanners` option as per the following example:

```
concat: {
  sample: {
    options: {
      stripBanners: true
    },
    files: {
      'all.js': ['one.js', 'two.js', 'three.js']
    }
  }
}
```

Adding a banner and footer to the concatenated result

Once the concatenated result is generated, it can often be useful to add a banner or footer that provides some extra information. The following example does just that by making use of the `banner` and `footer` options:

```
concat: {
  sample: {
    options: {
      banner: '/* The combination of three files */\n',
      footer: '\n/* No more files to be combined */'
    },
    files: {
      'all.js': ['one.js', 'two.js', 'three.js']
    }
  }
}
```

Running the task with the preceding example code should produce an `all.js` file with contents similar to the following:

```
/* The combination of three files */
var sample = 'Sample code from one.js.\n';
console.log(sample);
var sample = 'Sample code from two.js.\n';
console.log(sample);
var sample = 'Sample code from three.js.\n';
console.log(sample);
```

```
/* No more files to be combined */
```

Processing the source file's contents before concatenation

In case we'd like to alter the contents of the files we're joining, the `concat` task provides the option of doing so with code, before concatenating them. This can be done by providing a function to the `process` option, which receives the file content, modifies it, and returns it.

The following example makes use of this functionality to create a banner above each joined file that indicates its filename:

```
concat: {
  sample: {
    options: {
      process: function (content, filename) {
        var banner = '/* ' + filename + ' */\n';
        return banner + content;
      }
    },
    files: {
      'all.js': ['one.js', 'two.js', 'three.js']
    }
  }
}
```

Running the task with the preceding example code should produce an `all.js` file with contents similar to the following:

```
/* one.js */
var sample = 'Sample code from one.js.\n';
console.log(sample);
/* two.js */
var sample = 'Sample code from two.js.\n';
console.log(sample);
/* three.js */
var sample = 'Sample code from three.js.\n';
console.log(sample);
```

Fetching a single URL

If you're working on web-based projects, you'll probably want to download something at some point. Downloading a file is quite a simple operation, but automating it can save you a significant amount of time in the long run.

We'll make use of the relatively popular `curl (2.0.2)` plugin to download a resource from the Internet. The `curl` task is provided by this plugin to download single files.

Getting ready

In this example, we'll work with the basic project structure we created in the *Installing Grunt on a project* recipe in Chapter 1, *Getting Started with Grunt*. Be sure to refer to it if you are not yet familiar with its contents.

How to do it...

The following steps will take us through using the **OpenWeatherMap API** to download the current weather for the city of London and saving it to the `london.json` file.

1. We'll start by installing the package that contains the `curl` plugin as per the instructions provided in the *Installing a plugin* recipe in Chapter 1, *Getting Started with Grunt*.
2. Now, we can add the following configuration to download London's current weather information to the `london.json` file:

   ```
   curl: {
         london: {
           src: 'http://api.openweathermap.org/'
             + 'data/2.5/weather?q=London',
           dest: 'london.json'
         }
       }
   ```

3. To test our setup, we can run the `grunt curl` command, which should provide us with output similar to the following:

   ```
   Running "curl:london" (curl) task
   File "london.json" created.
   ```

4. If we now take a look at the project directory, we should see the downloaded weather information saved to the `london.json` file.

There's more...

If we need more flexibility in our requests to download content, the `curl` task allows us to do so using the `request` HTTP client utility provided by the `request` package. The same options that can be used for the `request` utility can be provided in the `src` configuration.

Tip

You can read more about the `request` HTTP client utility at the following URL:

https://github.com/mikeal/request

The following example uses this functionality to provide an object for the query string of the request, instead of having it included in the URL string:

```
curl: {
  london: {
    src: {
      url: 'http://api.openweathermap.org/data/2.5/weather',
      qs: {q:'London'}
    },
    dest: 'london.json'
  }
}
```

Fetching multiple URLs

If you're automating resource downloads for your project, you'll probably want to download more than one resource to the same directory, and probably from the same site.

We'll make use of the relatively popular `curl (2.0.2)` plugin to download multiple resources from the Internet. It provides the `curl-dir` task specifically to download multiple resources.

Getting ready

In this example, we'll work with the basic project structure we created in the *Installing Grunt on a project* recipe in Chapter 1, *Getting Started with Grunt*. Be sure to refer to it if you are not yet familiar with its contents.

How to do it...

The following steps take us through downloading both the Grunt and Node.js logos into the `logos` directory.

1. We'll start by installing the package that contains the `curl` plugin as per the instructions provided in the *Installing a plugin* recipe in Chapter 1, *Getting Started with Grunt*.
2. Now, we can add the following `curl-dir` task configuration, which will download the two logos into the `logos` directory:

```
'curl-dir': {
  logos: {
    src: [
      'http://nodejs.org/images/logo.png',
      'http://www.gruntjs.com/img/grunt-logo.png'
    ],
    dest: 'logos'
  }
}
```

3. To test the task, we can run it using the `grunt curl-dir` command, which should provide output similar to the following:

```
Running "curl-dir:logo" (curl-dir) task
Files "logos/logo.png", "logos/grunt-logo.png" created.
```

4. We should now have a directory called `logos` in our project path that contains both the Grunt and Node.js logos.

There's more...

The `curl-dir` task provides us with several options that allow us to download similar URLs, save downloaded files to altered filenames, and download files using special request options.

Downloading similar URLs

It's also quite common that we might want to download multiple URLs that are very similar. In this case, we can make use of the brace expansion support provided by the `curl-dir` task.

The following configuration example illustrates how we would go about downloading the weather information for London, Paris, and Tokyo, without having to repeat the start of each URL:

```
'curl-dir': {
  weather: {
    src: 'http://api.openweathermap.org/'
      + 'data/2.5/weather?q={London,Paris,Tokyo}',
    dest: 'weather'
  }
}
```

Saving downloaded files to altered filenames

It is also quite common to save downloaded resources to filenames other than the ones contained in their URLs. The `router` configuration provided by the `curl-dir` task allows us to specify a function that receives the URL we downloaded and return the filename that the resource should be saved to.

The following configuration example illustrates how we can download weather information for three cities and save each of the results to a file with only the city name and the `.json` file extension:

```
'curl-dir': {
  weather: {
    router: function (url) {
      var city = url.slice(url.indexOf('=') + 1, url.length);
      return city + '.json';
    },
    src: 'http://api.openweathermap.org/'
      + 'data/2.5/weather?q={London,Paris,Tokyo}',
    dest: 'weather'
  }
}
```

Downloading files using special request options

If we need more flexibility in our requests to download content, the `curl-dir` task allows us to do so using the `request` HTTP client utility provided by the `request` package. The same options that can be used for the `request` utility can be provided in the `src` configuration.

The following configuration example makes use of this functionality by providing the query parameters as objects rather than strings that are appended to the URLs, and also demonstrates using the `router` configuration discussed earlier for this type of setup:

```
'curl-dir': {
  weather: {
    router: function (src) {
      return src.qs.q + '.json';
    },
    src: [{
      url: 'http://api.openweathermap.org/data/2.5/weather',
      qs: {q:'London'}
    }, {
      url: 'http://api.openweathermap.org/data/2.5/weather',
      qs: {q:'Paris'}
    }],
    dest: 'weather'
  }
}
```

Chapter 3. Templating Engines

In this chapter, we will cover the following recipes:

- Rendering Jade templates
- Using data in a Jade template
- Using custom filters in a Jade template
- Compiling Jade templates
- Compiling Handlebars templates
- Compiling Underscore templates
- Using partials in Handlebars templates
- Wrapping Jade templates in AMD modules
- Wrapping Handlebars templates in AMD modules
- Wrapping Underscore templates in AMD modules
- Wrapping Handlebars templates in CommonJS modules
- Altering Jade templates before compilation
- Altering Handlebars templates before compilation
- Altering Underscore templates before compilation

Introduction

The generation of content by combining logic and data poses a unique set of problems, and template engines are specifically designed to solve them. In this chapter, we'll mostly be focusing on generating HTML using various template engines, even though some of them are designed in such a way as to allow for the generation of pretty much any type of readable file format.

Even though many of the more feature-rich web application frameworks available in the JavaScript community provide for the on-demand compilation, rendering and caching of templates, doing so using Grunt provides for the flexibility required in building a highly optimized and specialized solution. Templates can be rendered directly to HTML or compiled to JSTs that can be used directly without any extra processing. The products of these templates can also be packaged to be made available in pretty much any environment you might want to use them in.

At the time of writing, there are many great templating engines available, and the majority of them support Grunt by way of plugins, but in this chapter, we'll focus on the ones most widely used by the Grunt community.

Rendering Jade templates

The Jade template engine allows us to easily build and maintain HTML templates using its minimal yet familiar syntax. In this recipe, we'll make use of the `contrib-jade (0.12.0)` plugin to compile a template that renders a simple HTML page.

Getting ready

In this example, we'll work with the basic project structure we created in the *Installing Grunt on a project* recipe in Chapter 1, *Getting Started with Grunt*. Be sure to refer to it if you are not yet familiar with its contents.

How to do it...

The following steps will take us through creating a simple Jade template and rendering it to an HTML file.

1. We'll start by installing the package that contains the `contrib-jade` plugin as per the instructions provided in the *Installing a plugin* recipe in Chapter 1, *Getting Started with Grunt*.
2. Let's create a simple Jade template called `index.jade` in our project directory, and add the following contents:

   ```
   doctype html
   html
     head
       title Index
     body
       h1 Index
   ```

3. Now, we can add the following `jade` task to our configuration, which will compile the `index.jade` file we just created to the `index.html` file in our project directory:

   ```
   jade: {
     sample: {
       options: {
         pretty: true
       },
       src: 'index.jade',
       dest: 'index.html'
     }
   }
   ```

   ### Tip

   We set the `pretty` option to `true` in this example so that the HTML generated from the template can be more readable. This is usually preferred when your project is still in development, but is usually not used in production, as enabling this option increases the size of the generated files.

4. Finally, we can run the task using the `grunt jade` command, which should produce output similar to the following:

   ```
   Running "jade:sample" (jade) task
   File index.html created.
   ```

5. If we now take a look at our project directory, we should see the new `index.html` file with the following contents:

   ```
   <!DOCTYPE html>
   <html>
     <head>
       <title>Index</title>
     </head>
     <body>
       <h1>Index</h1>
     </body>
   </html>
   ```

Using data in a Jade template

Once we've got a Jade template, we can use it to render the same page structure with a variety of data. In this recipe, we'll make use of the `contrib-jade (0.12.0)` plugin in conjunction with the `data` option to send data that should be used in the rendering of the template.

Getting ready

In this example, we'll work with the basic project structure we created in the *Rendering Jade templates* recipe in this chapter. Be sure to refer to it if you are not yet familiar with its contents.

How to do it...

The following steps take us through providing data when rendering our template, and altering our `index.jade` template to make use of the provided data.

1. First, we'll alter our `index.jade` template to make use of the variables provided by its context:

   ```
   doctype html
   html
     head
       title= title
     body
       h1= title
       p= body
   ```

2. Now, we can provide values for the `title` and `body` variables that are to be used by the template. This is done by specifying them in the `data` option that we'll add to the configuration of our `jade` task:

   ```
   jade: {
     sample: {
       options: {
         pretty: true,
         data: {
           title: 'Most Amazing Site',
           body: 'Disappointing mundane content.'
         }
       },
       src: 'index.jade',
       dest: 'index.html'
     }
   }
   ```

3. Finally, we can run the task using the `grunt jade` command, which should produce output similar to the following:

   ```
   Running "jade:sample" (jade) task
   File index.html created.
   ```

4. If we now take a look at the `index.html` file that was generated by running the task, we'll find that the variables indicated in the template were replaced by the values specified in the `data` option:

   ```
   <!DOCTYPE html>
   <html>
   ```

```
  <head>
    <title>Most Amazing Site</title>
  </head>
  <body>
    <h1>Most Amazing Site</h1>
    <p>Disappointing mundane content.</p>
  </body>
</html>
```

There's more...

It's usually not a very good idea to keep the data used in a template hardcoded in the Grunt configuration, but rather to have it imported from an external data source.

The following steps take us through creating an external data source and altering our configuration so that the data used in rendering our template is loaded from it.

1. First, we'll create a file called `data.json` in our project directory, which contains the data we wish to use in the rendering of our template:

   ```
   {
     "title": "Most Amazing Site",
     "body": "Disappointing mundane content."
   }
   ```

2. Then, we can alter our `jade` task's configuration to import the data from the external source by making use of the `grunt.file.readJSON` method:

   ```
   jade: {
     sample: {
       options: {
         pretty: true,
         data: grunt.file.readJSON('data.json')
       },
       src: 'index.jade',
       dest: 'index.html'
     }
   }
   ```

3. If we now run the task using the `grunt jade` command, we should have the exact same result as in the main recipe.

Using custom filters in a Jade template

The filters provided by the Jade template engine enable us to indicate the method that should be used when processing a specific block in a template. This is usually used for blocks of content that are written in a format other than the Jade language itself. As an example, the `coffee` and `markdown` filters provided by the Jade library provide the rendering of **CoffeeScript** code into JavaScript and **Markdown** content into HTML.

In this recipe, we'll make use of the `contrib-jade (0.12.0)` plugin in conjunction with its `filters` option to make a custom filter called `link` available to us in our templates. This filter will use a simplistic algorithm to find URLs and surround them with an anchor tag, turning them into links.

Getting ready

In this example, we'll work with the basic project structure we created in the *Rendering Jade templates* recipe of this chapter. Be sure to refer to it if you are not yet familiar with its contents.

How to do it...

The following steps take us through adding the `link` custom filter and altering our `index.jade` template to make use of it.

1. First, we'll add the `link` custom filter to our configuration by using the `filters` option in the `jade` task we configured before:

```
jade: {
  sample: {
    options: {
      pretty: true,
      filters: {
        link: function (block) {
          var url_re = /(http:\/\/[^\s]*)/;
          var link_replace = '<a href="$1">$1</a>';
          return block.replace(url_re, link_replace);
        }
      }
    },
    src: 'index.jade',
    dest: 'index.html'
  }
}
```

2. Now, we can alter the contents of our `index.jade` template to make use of our `link` custom filter:

```
doctype html
html
  head
    title Index
  body
    h1 Index
    p
      :link
        Be sure to check out
        http://gruntjs.com/
        for more information.
```

3. Finally, we can run the task using the `grunt jade` command, which should produce output similar to the following:

```
Running "jade:sample" (jade) task
File index.html created.
```

4. If we now take a look at the `index.html` file that was generated by running the

task, we'll find that it has surrounded the link we had inside the block provided to the `link` filter with an anchor tag:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Index</title>
  </head>
  <body>
    <h1>Index</h1>
    <p>
      Be sure to check out
      <a href="http://gruntjs.com/">http://gruntjs.com/</a>
      for more information.
    </p>
  </body>
</html>
```

Compiling Jade templates

When building a web application that needs to render HTML templates on the frontend in as little time as possible, the compiling templates become essential. The Jade templating engine allows us to compile our templates to **JavaScript Templates** (**JSTs**) for use on the frontend.

In this recipe, we'll make use of the `contrib-jade (0.12.0)` plugin to compile a template that renders a minimalistic blog.

Getting ready

In this example, we'll work with the basic project structure we created in the *Installing Grunt on a project* recipe in Chapter 1, *Getting Started with Grunt*. Be sure to refer to it if you are not yet familiar with its contents.

How to do it...

The following steps take us through creating a simple Jade template and compiling it to a JST contained in another file.

1. We'll start by installing the package that contains the `contrib-jade` plugin as per the instructions provided in the *Installing a plugin* recipe in Chapter 1, *Getting Started with Grunt*.
2. Let's create a simple Jade template file called `blog.jade` in our project directory and give it the following contents:

   ```
   .blog
     h1= title
     each post in posts
       .post
         h2= post.title
         p= post.body
   ```

3. Now, we can add the following `jade` task to our configuration, which will compile the `blog.jade` template into a JST contained inside the `templates.js` file of our project directory:

   ```
   jade: {
     blog: {
       options: {
         client: true
       },
       src: 'blog.jade',
       dest: 'templates.js'
     }
   }
   ```

4. Finally, we can run the task using the `grunt jade` command, which should produce output similar to the following:

   ```
   Running "jade:blog" (jade) task
   File templates.js created.
   ```

5. We should now have a new file called `templates.js` in our project directory that contains the JST code for our compiled template.

How it works...

The following steps demonstrate the usage template of the compiled and display an example of the rendered result.

1. First, you'll need to include the Jade runtime library in the application or page in which you'd like to make use of the compiled templates.

   Tip

   At the time of writing, the Jade runtime library was available at the root of the official Jade repository and could be downloaded from the following URL:

   https://raw.githubusercontent.com/jadejs/jade/1.11.0/runtime.js

2. Then, you'll need to include the `templates.js` file that was generated by the `jade` task in the application or page to make the `JST` global variable that contains the templates available.
3. The following example code will render the compiled template, using some sample data and store the result in the `result` variable:

   ```
   var result = JST['blog']({
     title: 'My Awesome Blog',
     posts: [{
       title: 'First Post',
       body: 'My very first post.'
     }, {
       title: 'Second Post',
       body: 'This is getting old.'
     }]
   });
   ```

4. The `result` variable should now contain the following HTML:

   ```
   <div class="blog">
     <h1>My Awesome Blog</h1>
       <h2>First Post</h2>
       <p>My very first post.</p>
       <h2>Second Post</h2>
       <p>This is getting old.</p>
   </div>
   ```

There's more...

The `jade` task provides some useful options in conjunction with the basic compilation of templates that allows us to specify the namespace of compiled templates, indicate how template names should be derived from their filenames, and compile templates with debug support.

Specifying the namespace of compiled templates

By default, compiled templates are stored in the `JST` namespace, but this can be changed to anything we like, by using the `namespace` option. In the following example, we configure the task to store templates in the `Templates` namespace:

```
jade: {
  blog: {
    options: {
      client: true,
      namespace: 'Templates'
    },
    src: 'blog.jade',
    dest: 'templates.js'
  }
}
```

Indicating how template names should be derived from filenames

The `processName` option can be used to indicate how the names under which templates are to be stored in the namespace should be derived from their filenames. The default behavior of the `jade` task is to use everything before the file's extension. In the following example, we indicate that the entire filename should be in uppercase:

```
jade: {
  blog: {
    options: {
      client: true,
      processName: function (filename) {
        return filename.toUpperCase();
      }
    },
    src: 'blog.jade',
    dest: 'templates.js'
  }
}
```

Compiling templates with debug support

The Jade template engine provides extra debug support for compiled templates, which can be enabled by way of the `compileDebug` option, as it is in the following example:

```
jade: {
  blog: {
    options: {
      client: true,
      compileDebug: true
    },
    src: 'blog.jade',
    dest: 'templates.js'
  }
}
```

Compiling Handlebars templates

The Handlebars template engine simplifies the building and maintaining of any markup or other readable files. Its familiar syntax, extensibility, and logic-less approach provides a great all-round template building experience. The engine is focused mainly on compiling templates to JSTs that are mostly used in the frontend of a web application.

In this recipe, we'll make use of the `contrib-handlebars (0.8.0)` plugin to compile a template that renders a minimalistic blog.

Getting ready

In this example, we'll work with the basic project structure we created in the *Installing Grunt on a project* recipe in Chapter 1, *Getting Started with Grunt*. Be sure to refer to it if you are not yet familiar with its contents.

How to do it...

The following steps take us through creating a simple Handlebars template and setting up Grunt to compile it to a JST contained in another file.

1. We'll start by installing the package that contains the `contrib-handlebars` plugin as per the instructions provided in the *Installing a plugin* recipe in Chapter 1, *Getting Started with Grunt*.
2. Let's create a simple Handlebars template file called `blog.hbs` in our project directory and give it the following contents:

```
<div class="blog">
  <h1>{{title}}</h1>
  {{#posts}}
    <h2>{{title}}</h2>
    <p>{{body}}</p>
  {{/posts}}
</div>
```

3. Now, we can add the following `handlebars` task to our configuration, which will compile the `post.hbs` template into a JST contained inside the `templates.js` file:

```
handlebars: {
  blog: {
    src: 'blog.hbs',
    dest: 'templates.js'
  }
}
```

4. Finally, we can run the task using the `grunt handlebars` command, which should produce output similar to the following:

```
Running "handlebars:blog" (handlebars) task
>> 1 file created.
```

5. We should now have a new file called `templates.js` in our project directory that contains the JST code for our compiled template.

How it works...

The following steps demonstrate the usage of the compiled template and display an example of the rendered result.

1. First, you'll need to include the Handlebars runtime library in the application or page in which you'd like to make use of the compiled templates.

   Tip

   At the time of writing, the Handlebars runtime library could be downloaded from the following URL:

   http://builds.handlebarsjs.com.s3.amazonaws.com/handlebars-v1.3.0.js

2. Then, you'll need to include the `templates.js` file that was generated by the `handlebars` task in the application or page to create the `JST` global variable that contains the templates available.
3. The following example code will render the compiled template, using some sample data and store the result in the `result` variable:

```
var result = JST['blog.hbs']({
  title: 'My Awesome Blog',
  posts: [{
    title: 'First Post',
    body: 'My very first post.'
  }, {
    title: 'Second Post',
    body: 'This is getting old.'
  }]
});
```

4. The `result` variable should now contain the following HTML:

```
<div class="blog">
  <h1>My Awesome Blog</h1>
    <h2>First Post</h2>
    <p>My very first post.</p>
    <h2>Second Post</h2>
    <p>This is getting old.</p>
</div>
```

There's more...

The `handlebars` task provides some useful options in conjunction with the basic compilation of templates which allows us to specify the namespace for compiled templates and indicate how template names should be derived from their filenames.

Specifying the namespace for compiled templates

By default, compiled templates are stored in the `JST` namespace, but this can be changed to anything we like, by using the `namespace` option. In the following example, we configure the task to store templates in the `Templates` namespace:

```
handlebars: {
  blog: {
    options: {
      namespace: 'Templates'
    },
    src: 'blog.hbs',
    dest: 'templates.js'
  }
}
```

Indicating how template names should be derived from file names

The `processName` option can be used to indicate how the names under which templates are to be stored in the namespace should be derived from their file names. The default behavior of the `handlebars` task is to use the entire filename. In the following example, we indicate that it should still use the file name, but in uppercase letters:

```
handlebars: {
  blog: {
    options: {
      processName: function (filename) {
        return filename.toUpperCase();
      }
    },
    src: 'blog.hbs',
    dest: 'templates.js'
  }
}
```

Compiling Underscore templates

Along with the many utilities it provides, the Underscore library also includes a simple, fast, and flexible templating engine. These templates are most commonly compiled to JSTs for use in the frontend of a web application.

In this recipe, we'll make use of the `contrib-jst (0.6.0)` plugin to compile a template that renders a minimalistic blog.

Getting ready

In this example, we'll work with the basic project structure we created in the *Installing Grunt on a project* recipe in Chapter 1, *Getting Started with Grunt*. Be sure to refer to it if you are not yet familiar with its contents.

How to do it...

The following steps take us through creating a simple Underscore template and compiling it to a JST contained in another file.

1. We'll start by installing the package that contains the `contrib-jst` plugin as per the instructions provided in the *Installing a plugin* recipe in Chapter 1, *Getting Started with Grunt*.
2. Let's create a simple Underscore template file called `blog.html` in our project directory and give it the following contents:

```
<div class="blog">
  <h1><%= title %></h1>
  <% _.each(posts, function (post) { %>
    <div class="post">
      <h2><%= post.title %></h2>
      <p><%= post.body %></p>
    </div>
  <% }) %>
</div>
```

3. Now, we can add the following `jst` task to our configuration, which will compile the `blog.html` template into a JST contained inside the `templates.js` file in our project directory:

```
jst: {
  blog: {
    src: 'blog.html',
    dest: 'templates.js'
  }
}
```

4. Finally, we can run the task using the `grunt jst` command, which should produce output similar to the following:

```
Running "jst:blog" (handlebars) task
>> 1 file created.
```

5. We should now have a new file called `templates.js` in our project directory, which contains the JST code for our compiled template.

How it works...

The following steps demonstrate the usage of the compiled template and display an example of the rendered result.

1. First, you'll need to include the Underscore runtime library in the application or page in which you'd like to make use of the compiled templates.

   ### Tip

   At the time of writing, the Underscore runtime library could be downloaded from the following URL:

   http://underscorejs.org/underscore-min.js

2. Then, you'll need to include the `templates.js` file that was generated by the `jst` task in the application or page to make the `JST` global variable that contains the templates available.
3. The following example code will render the compiled template, using some sample data and store the result in the `result` variable:

```
var result = JST['blog.html']({
  title: 'My Awesome Blog',
  posts: [{
    title: 'First Post',
    body: 'My very first post.'
  }, {
    title: 'Second Post',
    body: 'This is getting old.'
  }]
});
```

4. The `result` variable should now contain the following HTML:

```
<div class="blog">
  <h1>My Awesome Blog</h1>
    <h2>First Post</h2>
    <p>My very first post.</p>
    <h2>Second Post</h2>
    <p>This is getting old.</p>
</div>
```

There's more...

The `jst` task provides some useful options in conjunction with the basic compilation of templates which allows us to specify the namespace for compiled templates and indicate how template names should be derived from their file names.

Specifying the namespace for compiled templates

By default, compiled templates are stored in the `JST` namespace, but this can be changed to anything we like, by using the `namespace` option. In the following example, we configure the task to store templates in the `Templates` namespace:

```
jst: {
  blog: {
    options: {
      namespace: 'Templates'
    },
    src: 'blog.html',
    dest: 'templates.js'
  }
}
```

Indicating how template names should be derived from file names

The `processName` option can be used to indicate how the names under which templates are to be stored in the namespace should be derived from their file names. The default behavior of the `jst` task is to use the entire file name. In the following example, we indicate that it should still use the file name, but in uppercase letters:

```
jst: {
  blog: {
    options: {
      processName: function (filename) {
        return filename.toUpperCase();
      }
    },
    src: 'blog.html',
    dest: 'templates.js'
  }
}
```

Using partials in Handlebars templates

The partials system provided by the Handlebars templating engine allows us to reuse smaller bits of template code in varying contexts. Whenever you see a pattern repeated at different points in your template, it's probably a good opportunity to make use of a partial.

In this recipe, we'll make use of the `contrib-handlebars (0.8.0)` plugin and its partial template loading functionality to render posts with a similar structure in two different sections of a blog.

Getting ready

In this example, we'll work with the basic project structure we created in the *Compiling Handlebars templates* recipe of this chapter. Be sure to refer to it if you are not yet familiar with its contents.

How to do it...

The following steps take us through creating a post partial template and altering our blog template so that it renders a new and old section.

1. By default, the `handlebars` task identifies a template starting with an underscore as a partial template and loads it. Let's create a file called `_post.hbs` in our project directory, which will contain our partial post template:

   ```
   <div class="post">
     <h3>{{title}}</h3>
     <p>{{body}}</p>
   </div>
   ```

2. Now, we'll alter the configuration of the task by adding the new partial template's filename to the list of templates that should be included in the task:

   ```
   handlebars: {
     sample: {
       src: ['blog.hbs', '_post.hbs'],
       dest: 'templates.js'
     }
   }
   ```

3. Once the partial template is set up to be included, we can alter our `blog.hbs` template to make use of it. Let's change it so that it renders a new and old section:

   ```
   <div class="blog">
     <h1>{{title}}</h1>
     <h2>New Posts</h2>
     {{#newPosts}}
       {{>post}}
     {{/newPosts}}
     <h2>Old Posts</h2>
     {{#oldPosts}}
       {{>post}}
     {{/oldPosts}}
   </div>
   ```

4. Finally, we can run the task using the `grunt handlebars` command, which should produce output similar to the following:

   ```
   Running "handlebars:blog" (handlebars) task
   >> 1 file created.
   ```

5. We should now have a modified version of the `templates.js` file in our project

directory, which contains the JST code for our latest blog template.

How it works...

The following steps demonstrate the usage of the compiled template and display an example of the rendered result.

1. First, you'll need to include the Handlebars runtime library in the application or page in which you'd like to make use of the compiled templates.

   ### Tip

   At the time of writing, the Handlebars runtime library could be downloaded from the following URL:

   http://builds.handlebarsjs.com.s3.amazonaws.com/handlebars-v1.3.0.js

2. Then, you'll need to include the `templates.js` file that was generated by the `handlebars` task in the application or page to make the `JST` global variable that contains the templates available.
3. The following example code will render the compiled template, using some sample data and store the result in the `result` variable:

```
var result = JST['blog.hbs']({
  title: 'My Awesome Blog',
  newPosts: [{
    title: 'Brand Spanking',
    body: 'Still has that new post smell.'
  }, {
    title: 'Shiny Post',
    body: 'Just got off the floor.'
  }],
  oldPosts: [{
    title: 'Old News',
    body: 'Not that relevant anymore.'
  }, {
    title: 'Way Back When',
    body: 'My very first post.'
  }]
});
```

4. The `result` variable should now contain the following HTML:

```
<div class="blog">
  <h1>My Awesome Blog</h1>
  <h2>New Posts</h2>
  <div class="post">
    <h3>Brand Spanking</h3>
    <p>Still has that new post smell.</p>
  </div>
  <div class="post">
    <h3>Shiny Post</h3>
    <p>Just got off the floor.</p>
  </div>
  <h2>Old Posts</h2>
  <div class="post">
    <h3>Old News</h3>
    <p>Not that relevant anymore.</p>
  </div>
  <div class="post">
    <h3>Way Back When</h3>
    <p>My very first post.</p>
  </div>
</div>
```

There's more...

The handlebars task provides some useful options in conjunction with the loading of partial templates that allow us to make partials available in the namespace, indicate how partial templates should be identified, and also indicate how partial template names should be derived.

Making partials available in the namespace

In the case that we'd like to make use of partial templates directly in our code and not just in our other templates, we can make them available in the namespace along with the other templates by using the partialsUseNamespace option as we do in the following example:

```
handlebars: {
  blog: {
    options: {
      partialsUseNamespace: true
    },
    src: ['blog.hbs', '_post.hbs'],
    dest: 'templates.js'
  }
}
```

Indicating how partial templates should be identified

We'll probably reach the point where we would like to change the way partial templates are identified. This can be done using a combination of the partialRegex and partialsPathRegex options. The following example indicates that all templates found in the partials directory should be loaded as partials:

```
handlebars: {
  blog: {
    options: {
      partialRegex: /.*/,
      partialsPathRegex: /partials\//
    },
    src: ['blog.hbs', 'partials/post.hbs'],
    dest: 'templates.js'
  }
}
```

Indicating how partial template names should be derived

By default, the `handlebars` task removes the preceding underscore and file extension from partial file names to determine their names. The following example makes use of the `processPartialName` option to indicate that the entire file name should be uppercased before use:

```
handlebars: {
  blog: {
    options: {
      processPartialName: function (filename) {
        return filename.toUpperCase();
      }
    },
    src: ['blog.hbs', '_post.hbs'],
    dest: 'templates.js'
  }
}
```

Wrapping Jade templates in AMD modules

In this recipe, we'll make use of the `contrib-jade (0.12.0)` plugin and its `amd` option to wrap our compiled blog template in an **Asynchronous Module Definition** (**AMD**) module.

Getting ready

In this example, we'll work with the basic project structure we created in the *Compiling Jade templates* recipe of this chapter. Be sure to refer to it if you are not yet familiar with its contents.

How to do it...

The following steps take us through altering our configuration so that it wraps our compiled templates in an AMD module.

1. First, we'll alter the configuration by adding the `amd` option to indicate that the compiled templates should be wrapped in an AMD module:

   ```
   jade: {
     blog: {
       options: {
         amd: true,
         client: true
       },
       src: 'blog.jade',
       dest: 'templates.js'
     }
   }
   ```

2. Using the `namespace` option, we can now also specify that we no longer want the compiled templates to be contained within the `JST` namespace, since they will now be contained within their own module:

   ```
   jade: {
     blog: {
       options: {
         amd: true,
         client: true,
         namespace: false
       },
       src: 'blog.jade',
       dest: 'templates.js'
     }
   }
   ```

3. We can now also compile the template to a different file name, as each template will now be represented by its own file:

   ```
   jade: {
     blog: {
       options: {
         amd: true,
         client: true,
         namespace: false
       },
       src: 'blog.jade',
       dest: 'blog.js'
   ```

```
    }
  }
```

4. Finally, we can run the task using the `grunt jade` command, which should produce output similar to the following:

```
Running "jade:blog" (jade) task
File blog.js created.
```

5. There should now be a `blog.js` file located in the project directory that contains the compiled template wrapped in an AMD module.

How it works...

The following steps demonstrate the usage of the compiled and wrapped template and display an example of the rendered result.

1. First, you'll need to ensure that the AMD framework you have in place has access to the Jade runtime library under the name `jade`. This may require the configuration of the AMD framework to be altered.

   ### Tip

   At the time of writing, the Jade runtime library was available in the root of the official Jade repository and could be downloaded from the following URL:

   https://raw.githubusercontent.com/jadejs/jade/1.11.0/runtime.js

2. Then, you'll need to make sure that the `blog.js` file that was generated by the `jade` task is accessible to the AMD framework under whichever name or path you prefer.
3. The following example code will fetch the compiled template as a dependency, render the compiled template using some sample data, and then proceed to store the result in the `result` variable:

```
define(['templates/blog'], function(blog) {
  var result = blog({
    title: 'My Awesome Blog',
    posts: [{
      title: 'First Post',
      body: 'My very first post.'
    }, {
      title: 'Second Post',
      body: 'This is getting old.'
    }]
  });
});
```

4. The `result` variable should now contain the following HTML:

```
<div class="blog">
  <h1>My Awesome Blog</h1>
    <h2>First Post</h2>
    <p>My very first post.</p>
    <h2>Second Post</h2>
    <p>This is getting old.</p>
</div>
```

Wrapping Handlebars templates in AMD modules

In this recipe, we'll make use of the `contrib-handlebars (0.8.0)` plugin and its `amd` option to wrap our compiled blog template in an AMD module.

Getting ready

In this example, we'll work with the basic project structure we created in the *Compiling Handlebars templates* recipe of this chapter. Be sure to refer to it if you are not yet familiar with its contents.

How to do it...

The following steps take us through altering our configuration so that it wraps our compiled templates in an AMD module.

1. First, we'll alter the configuration by adding the `amd` option to indicate that the compiled templates should be wrapped in an AMD module:

   ```
   handlebars: {
     blog: {
       options: {
         amd: true
       },
       src: 'blog.hbs',
       dest: 'templates.js'
     }
   }
   ```

2. Using the `namespace` option, we can now also specify that we no longer want the compiled templates to be contained within the `JST` namespace, since they will now be contained within their own module:

   ```
   handlebars: {
     blog: {
       options: {
         amd: true,
         namespace: false
       },
       src: 'blog.hbs',
       dest: 'templates.js'
     }
   }
   ```

3. We can now also compile the template to a different file name, as each template will now be represented by its own file:

   ```
   handlebars: {
     blog: {
       options: {
         amd: true,
         namespace: false
       },
       src: 'blog.hbs',
       dest: 'blog.js'
     }
   }
   ```

4. Finally, we can run the task using the `grunt handlebars` command, which should produce output similar to the following:

   ```
   Running "handlebars:blog" (handlebars) task
   >> 1 file created.
   ```

5. There should now be a `blog.js` file located in the project directory, which contains the compiled template wrapped in an AMD module.

How it works...

The following steps demonstrate the usage of the compiled and wrapped template and display an example of the rendered result.

1. First, you'll need to ensure that the AMD framework you have in place has access to the Handlebars runtime library under the `handlebars` name. This may require the configuration of the AMD framework to be altered.

   ### Tip

   At the time of writing, the Handlebars runtime library could be downloaded from the following URL:

   http://builds.handlebarsjs.com.s3.amazonaws.com/handlebars-v1.3.0.js

   In the case of the runtime library mentioned in this tip, you will have to make use of a shim as the library is not prepared for use with AMD modules. You can read more about shimming using the RequireJS framework at the following URL:

   http://requirejs.org/docs/api.html#config-shim

2. Then, you'll need to make sure that the `blog.js` file that was generated by the `handlebars` task is accessible to the AMD framework under whichever name or path you prefer.
3. The following example code will fetch the compiled template as a dependency, render the compiled template using some sample data, and then proceed to store the result in the `result` variable:

```
define(['templates/blog'], function(blog) {
  var result = blog({
    title: 'My Awesome Blog',
    posts: [{
      title: 'First Post',
      body: 'My very first post.'
    }, {
      title: 'Second Post',
      body: 'This is getting old.'
    }]
  });
});
```

4. The `result` variable should now contain the following HTML:

```
<div class="blog">
```

```
<div class="blog">
  <h1>My Awesome Blog</h1>
    <h2>First Post</h2>
    <p>My very first post.</p>
    <h2>Second Post</h2>
    <p>This is getting old.</p>
</div>
```

Wrapping Underscore templates in AMD modules

In this recipe, we'll make use of the `contrib-jst (0.6.0)` plugin and its `amd` option to wrap our compiled blog template in an AMD module.

Getting ready

In this example, we'll work with the basic project structure we created in the *Compiling Underscore templates* recipe of this chapter. Be sure to refer to it if you are not yet familiar with its contents.

How to do it...

The following steps take us through altering our configuration so that it wraps our compiled template in an AMD module.

1. First, we'll alter the configuration by adding the `amd` option to indicate that the compiled templates should be wrapped in an AMD module:

   ```
   jst: {
     blog: {
       options: {
         amd: true
       },
       src: 'blog.html',
       dest: 'templates.js'
     }
   }
   ```

2. Using the `namespace` option, we can now also specify that we no longer want the compiled templates to be contained within the `JST` namespace, since they will now be contained within their own module:

   ```
   jst: {
     blog: {
       options: {
         amd: true,
         namespace: false
       },
       src: 'blog.html',
       dest: 'templates.js'
     }
   }
   ```

3. We can now also compile the template to a different filename, as each template will now be located inside its own file:

   ```
   jst: {
     blog: {
       options: {
         amd: true,
         namespace: false
       },
       src: 'blog.html',
       dest: 'blog.js'
     }
   }
   ```

4. Finally, we can run the task using the `grunt jst` command, which should produce output similar to the following:

   ```
   Running "jst:blog" (jst) task
   File blog.js created.
   ```

5. There should now be a `blog.js` file located in the project directory that contains the compiled template wrapped in an AMD module.

How it works...

The following steps demonstrate the usage of the compiled and wrapped template and display an example of the rendered result.

1. First, you'll need to ensure that the AMD framework you have in place loads the Underscore library on startup, since the wrappers created by this plugin don't reference it as a dependency.

   ### Tip

   At the time of writing, the Underscore runtime library could be downloaded from the following URL:

   http://underscorejs.org/underscore-min.js

   You can read more about specifying startup dependencies on the RequireJS framework at the following URL:

   http://requirejs.org/docs/api.html#config-deps
2. Then, you'll need to make sure that the `blog.js` file that was generated by the `jst` task is accessible to the AMD framework under whichever name or path you prefer.
3. The following example code will fetch the compiled template as a dependency, render the compiled template using some sample data, and then proceed to store the result in the `result` variable:

   ```
   define(['templates/blog'], function(blog) {
     var result = blog({
       title: 'My Awesome Blog',
       posts: [{
         title: 'First Post',
         body: 'My very first post.'
       }, {
         title: 'Second Post',
         body: 'This is getting old.'
       }]
     });
   });
   ```
4. The `result` variable should now contain the following HTML:

   ```
   <div class="blog">
   ```

```
  <h1>My Awesome Blog</h1>
    <h2>First Post</h2>
    <p>My very first post.</p>
    <h2>Second Post</h2>
    <p>This is getting old.</p>
</div>
```

Wrapping Handlebars templates in CommonJS modules

In this recipe, we'll make use of the `contrib-handlebars (0.8.0)` plugin to wrap our compiled blog template in a **CommonJS** module.

Getting ready

In this example, we'll work with the basic project structure we created in the *Compiling Handlebars templates* recipe of this chapter. Be sure to refer to it if you are not yet familiar with its contents.

How to do it...

The following steps take us through altering our configuration so that it wraps our compiled template in a CommonJS module.

1. First, we'll alter the configuration by adding the `amd` option to indicate that the compiled templates should be wrapped in a CommonJS module.

   ```
   handlebars: {
     blog: {
       options: {
         commonjs: true
       },
       src: 'blog.hbs',
       dest: 'templates.js'
     }
   }
   ```

2. Using the `namespace` option, we can now also specify that we no longer want the compiled templates to be contained within the `JST` namespace, since they will now be contained within a module:

   ```
   handlebars: {
     blog: {
       options: {
         commonjs: true,
         namespace: false
       },
       src: 'blog.hbs',
       dest: 'templates.js'
     }
   }
   ```

3. Finally, we can run the task using the `grunt handlebars` command, which should produce output similar to the following:

   ```
   Running "handlebars:blog" (handlebars) task
   >> 1 file created.
   ```

4. There should now be a `templates.js` file located in the project directory that contains the compiled template wrapped in a CommonJS module.

How it works...

The following steps demonstrate the usage of the compiled and wrapped template and display an example of the rendered result.

1. First, we'll have to ensure that the Handlebars library is installed in the current CommonJS supporting environment. Using Node.js as an example, the module will have to be installed using the `npm install handlebars` command.
2. Next, we will have to make it available in the piece of code that we'll be importing the templates into, by using code similar to the following:

   ```
   var Handlebars = require('handlebars');
   ```

3. Then, we'll need to import the templates into our code and pass it to the Handlebars library by using code similar to the following:

   ```
   var templates = require('./templates')(Handlebars);
   ```

   ### Tip

   This bit of code assumes that the generated `templates.js` file is on the same path as the file containing the piece of code we're currently working on.

4. Now that we've imported the templates, we can make use of them. The following example renders the `blog.hbs` templates and stores the results in the `result` variable:

   ```
   var result = templates['blog.hbs']({
     title: 'My Awesome Blog',
     posts: [{
       title: 'First Post',
       body: 'My very first post.'
     }, {
       title: 'Second Post',
       body: 'This is getting old.'
     }]
   });
   ```

5. The `result` variable should now contain the following HTML:

   ```
   <div class="blog">
     <h1>My Awesome Blog</h1>
       <h2>First Post</h2>
       <p>My very first post.</p>
       <h2>Second Post</h2>
       <p>This is getting old.</p>
   ```

</div>

Altering Jade templates before compilation

There might come a time that you would want to alter the contents of a template before compiling or rendering it. This is most commonly required when you would like to remove excessive white space from templates, or if you'd like to remove or replace parts of a template you have no control over.

In this recipe, we'll make use of the `contrib-jade (0.12.0)` plugin and the `processContent` option provided by its `jade` task to remove whitespace from our templates before rendering them to HTML.

Getting ready

In this example, we'll work with the basic project structure we created in the *Compiling Jade templates* recipe of this chapter. Be sure to refer to it if you are not yet familiar with its contents.

How to do it...

The following steps take us through altering our configuration so that excessive whitespace is removed from our templates before rendering them to HTML.

1. First, we can provide an empty function to the `processContent` option that receives the content of the template and passes it through without altering it:

   ```
   jade: {
     sample: {
       options: {
         processContent: function (content) {
           return content;
         }
       },
       src: 'index.jade',
       dest: 'index.html'
     }
   }
   ```

2. Then, we can alter the code of the function provided to `processContent` so that it removes all of the trailing whitespace from the template's content:

   ```
   processContent: function (content) {
     content = content.replace(/[\x20\t]+$/mg, '');
     content = content.replace(/[\r\n]*$/, '\n');
     return content;
   }
   ```

3. If we now run the task using the `grunt jade` command, we will see output similar to the following:

   ```
   Running "jade:sample" (jade) task
   File index.html created.
   ```

4. The rendered result in the `index.html` file should now display no signs of any of the trailing whitespace that was present in the original template.

Altering Handlebars templates before compilation

Although not very common, there might come a time that you would want to alter the contents of a template before compiling or rendering it. This is most commonly required when you would like to remove excessive whitespace from templates, or if you'd like to remove or replace certain parts of a template you have no control over.

In this recipe, we'll make use of the `contrib-handlebars (0.8.0)` plugin and the `processContent` option provided by its `handlebars` task to remove whitespace from our templates before compiling them.

Getting ready

In this example, we'll work with the basic project structure we created in the *Compiling Handlebars templates* recipe of this chapter. Be sure to refer to it if you are not yet familiar with its contents.

How to do it...

The following steps take us through altering our configuration so that excessive whitespace is removed from our templates before compiling them.

1. First, we can provide an empty function to the `processContent` option that receives the content of the template and passes it through without altering it:

   ```
   handlebars: {
     blog: {
       options: {
         processContent: function (content) {
           return content;
         }
       },
       src: 'blog.hbs',
       dest: 'templates.js'
     }
   }
   ```

2. Then, we can alter the code of the function provided to `processContent` so that it removes all of the leading and trailing whitespace from the template's content:

   ```
   processContent: function (content) {
     content = content.replace(/^[\x20\t]+/mg, '');
     content = content.replace(/[\x20\t]+$/mg, '');
     content = content.replace(/^[\r\n]+/, '');
     content = content.replace(/[\r\n]*$/, '\n');
     return content;
   }
   ```

3. If we now run the task using the `grunt handlebars` command, we should see output similar to the following:

   ```
   Running "handlebars:blog" (handlebars) task
   >> 1 file created.
   ```

4. The compiled result in the `templates.js` file should now display no signs of any of the trailing whitespace that was present in the original template.

Altering Underscore templates before compilation

Although not very common, there might come a time that you would want to alter the contents of a template before compiling or rendering it. This is most commonly required when you would like to remove excessive whitespace from templates, or if you'd like to remove or replace certain parts of a template you have no control over.

In this recipe, we'll make use of the `contrib-jst (0.6.0)` plugin and the `processContent` option provided by its `jst` task to remove whitespace from our templates before compiling them.

Getting ready

In this example, we'll work with the basic project structure we created in the *Compiling Underscore templates* recipe of this chapter. Be sure to refer to it if you are not yet familiar with its contents.

How to do it...

The following steps take us through altering our configuration so that excessive whitespace is removed from our templates before compiling them.

1. First, we can provide an empty function to the `processContent` option that receives the content of the template and passes it through without altering it:

```
jst: {
  blog: {
    options: {
      processContent: function (content) {
        return content;
      }
    },
    src: 'blog.html',
    dest: 'templates.js'
  }
}
```

2. Then, we can alter the code of the function provided to `processContent` so that it removes all leading and trailing whitespace from the template's content:

```
processContent: function (content) {
  content = content.replace(/^[\x20\t]+/mg, '');
  content = content.replace(/[\x20\t]+$/mg, '');
  content = content.replace(/^[\r\n]+/, '');
  content = content.replace(/[\r\n]*$/, '\n');
  return content;
}
```

3. If we now run the task using the `grunt jst` command, we should see output similar to the following:

```
Running "jst:blog" (jst) task
File templates.js created.
```

4. The compiled result in the `templates.js` file should now display no signs of any of the trailing whitespace that was present in the original template.

Chapter 4. Generating CSS and JavaScript

In this chapter, we will cover the following recipes:

- Compiling LESS to CSS
- Compiling Sass to CSS
- Compiling Stylus to CSS
- Compiling CoffeeScript to JavaScript
- Compiling LiveScript to JavaScript
- Generating source maps for LESS
- Generating source maps for Sass
- Generating source maps for CoffeeScript
- Defining custom functions with LESS
- Using Stylus plugins

Introduction

Being as most of the technology in use today is based on standards, most notably the ones maintained by the W3C, and the evolution of standards being quite slow, there has always been a community of developers that aim to force evolution by building tools on top of existing antiquated technologies.

Generating **CSS** and **JavaScript** code has been around for quite some time now, and has become somewhat of an industry standard. Using the latest tools to generate code allows us to manage and build our web applications faster than ever before. We can now make use of more modern language features in our code, and we can finally make use of our programming skills to generate our CSS style sheets.

Compiling LESS to CSS

In this recipe, we'll make use of the `contrib-less (0.11.4)` plugin to compile our **LESS** style sheets to **CSS** style sheets, which can be used by our web applications.

Getting ready

In this example, we'll work with the basic project structure we created in the *Installing Grunt on a project* recipe in Chapter 1, *Getting Started with Grunt*. Be sure to refer to it if you are not yet familiar with its contents.

How to do it...

The following steps will take us through creating a simple LESS style sheet and compiling it to CSS:

1. We'll start by installing the package that contains the `contrib-less` plugin, as per the instructions provided in the *Installing a plugin* recipe in Chapter 1, *Getting Started with Grunt*.
2. Let's create a simple LESS file called `styles.less` in our project directory with the following contents:

   ```
   @first: #ffffff;
   @second: #000000;

   body {
     background-color: @first;
     background-image: url('../background.png');
     color: @second;
   }
   ```
3. Now, we can add the following `less` task to our configuration, which will compile the `styles.less` file to the `styles.css` file in our project directory:

   ```
   less: {
     styles: {
       src: 'styles.less',
       dest: 'styles.css'
     }
   }
   ```
4. Then, we can run the task by using the `grunt less` command, which will produce output similar to the following:

   ```
   Running "less:styles" (less) task
   File styles.css created: 0 B → 56 B
   ```
5. If we now take a look at our project directory, we should see the new `styles.css` file with the following contents:

   ```
   body {
     background-color: #ffffff;
     background-image: url('background.png');
     color: #000000;
   }
   ```

There's more...

The `less` task provides us with several useful options that can be used in conjunction with its basic compilation feature. We'll look at how to specify a root path for resources, minify the resulting output, rewrite URLs to be relative, force evaluation of imports, specify extra import paths, and add a banner to the resulting output.

Specifying a root path for resources

In case we'd like to prepend a path to all the URL resources in our L file, we can make use of the `rootpath` option, as per the following example:

```
less: {
  styles: {
    options: {
      rootpath: '../'
    },
    src: 'styles.less',
    dest: 'styles.css'
  }
}
```

Minifying the resulting output

As we're always working toward a smaller download size for our web applications, we will eventually reach the point where we'd like to reduce the size of our style sheets by minimizing them. The `cleancss` option can be used to indicate that the **clean-css** utility should be used to compress the compiled result, as per the following example:

```
less: {
  styles: {
    options: {
      cleancss: true
    },
    src: 'styles.less',
    dest: 'styles.css'
  }
}
```

Rewriting URLs to be relative

In case we import one LESS style sheet into another, the URL references to assets may become a bit confusing because we would have to reference resources into our style sheet, using URLs that are relative to whichever style sheet that might want to import it.

This would, of course, also break if we were importing the same style sheet into two different style sheets, each with a different location.

By enabling the `rewriteUrls` option, we don't have to worry about having incorrect resource references when importing style sheets. The compiler will check all the URLs of referenced resources in the style sheet that is being imported, and alter them to be relative to their new location.

Say we had the following directory structure with a main `style.less` file that imports the `body.less` file, which references the `background.png` image in the project root:

```
project/
├── background.png
├── styles.less
└── lib
    └── body.less
```

With the `rewriteUrls` option set to `false`, a reference from the `body.less` file of `url('../background.png')` will remain the same in the resulting `style.css` file.

With `rewriteUrls` set to `true`, as per the following example, it will be changed to `url('background.png')` as this will be the new correct relative path from the resulting `style.css` file to the `background.png` file:

```
less: {
  styles: {
    options: {
      relativeUrls: true
    },
    src: 'styles.less',
    dest: 'styles.css'
  }
}
```

Forcing evaluation of imports

In order to receive error messages for missing import dependencies, we can set the `strictImports` option to `true`, as per the following example:

```
less: {
  styles: {
    options: {
      strictImports: true
    },
    src: 'styles.less',
    dest: 'styles.css'
  }
}
```

Specifying extra import paths

In case we'd like to allow our LESS style sheets to import style sheets other than the ones available to them via a simple relative path, we can indicate extra directories that should be scanned when importing files by using the `paths` option. The following example indicates that the `lib` directory in our project root should be included when scanning for imports:

```
less: {
  styles: {
    options: {
      paths: ['lib']
    },
    src: 'styles.less',
    dest: 'styles.css'
  }
}
```

Adding a banner to the resulting output

In case we would like to add some extra information to the beginning of our compiled style sheet, we can make use of the `banner` option. The following example adds a banner that contains some information about the file we're producing:

```
less: {
  styles: {
    options: {
      banner: '/* Name: style.css */\n' +
        '/* Author: Grunt Wiz */\n' +
        '/* License: MIT */\n\n'

    },
    src: 'styles.less',
    dest: 'styles.css'
  }
}
```

Compiling Sass to CSS

In this recipe, we'll make use of the `sass (0.12.1)` plugin to compile our **Sass** style sheets to **CSS** style sheets, which can be used by our web application.

Tip

The contrib-sass plugin is slightly more popular and more mature that the sass plugin but requires the Sass **Ruby** library to be installed, which seems to be an unnecessary step for this recipe.

Please note that at the time of writing, the sass plugin did not work with Version `0.11.*` of Node.js, but a solution for this issue was in the works. In the following recipe, Version `0.10.*` was used.

Getting ready

In this example, we'll work with the basic project structure that we created in the *Installing Grunt on a project* recipe in Chapter 1, *Getting Started with Grunt*. Be sure to refer to it if you are not yet familiar with its contents.

How to do it...

The following steps will take us through creating a simple Sass style sheet and compiling it to CSS:

1. We'll start by installing the package that contains the `sass` plugin as per the instructions provided in the *Installing a plugin* recipe in Chapter 1, *Getting Started with Grunt*.
2. Let's create a simple Sass file called `styles.scss` in our project directory with the following contents:

   ```
   $first: #ffffff;
   $second: #000000;

   body {
     background-color: $first;
     background-image: url('background.png');
     color: $second;
   }
   ```

3. Now, we can add the following `sass` task to our configuration, which will compile the `styles.scss` file to the `styles.css` file in our project directory:

   ```
   sass: {
     styles: {
       src: 'styles.scss',
       dest: 'styles.css'
     }
   }
   ```

4. Then, we can run the task by using the `grunt sass` command, which will produce output similar to the following:

   ```
   Running "sass:styles" (sass) task
   File styles.css created.
   ```

5. If we now take a look at our project directory, we should see the new `styles.css` file with the following contents:

   ```
   body {
     background-color: #ffffff;
     background-image: url('background.png');
     color: #000000;
   }
   ```

There's more...

The `sass` task provides us with several useful options that can be used in conjunction with its basic compilation feature. We'll look at how to change the output style, specify a path for images, and specify extra import paths.

Changing the output style

There are two output styles available when compiling Sass style sheets, each providing a different level of readability. By default, the `outputStyle` option is set to `nested`, which produces a more readable output. To remove all white spaces from the output, you can set it to `compressed`, as per the following example:

```
sass: {
  styles: {
    options: {
      outputStyle: 'compressed'
    },
    src: 'styles.scss',
    dest: 'styles.css'
  }
}
```

Specifying a path for images

The `image-url` function is provided by the `sass` plugin if you'd like to have the ability to easily alter the paths to all the images referenced in a Sass style sheet. The following example makes use of the `imagePath` option to indicate that the `img` string should be prepended to every URL specified, using the `image-url` function:

```
sass: {
  styles: {
    options: {
      imagePath: 'img'
    },
    src: 'styles.scss',
    dest: 'styles.css'
  }
}
```

Specifying extra import paths

In case we'd like to allow our Sass style sheets to import style sheets other than the ones available to them via a simple relative path, we can indicate extra directories that should be scanned when importing files by using the `includePaths` option. The following example indicates that the `lib` directory in our project root should be included when scanning for imports:

```
sass: {
  styles: {
    options: {
      includePaths: ['lib']
    },
    src: 'styles.scss',
    dest: 'styles.css'
  }
}
```

Compiling Stylus to CSS

In this recipe, we'll make use of the `contrib-stylus (0.18.0)` plugin to compile our **Stylus** style sheets to **CSS** style sheets, which can be used by our web application.

Getting ready

In this example, we'll work with the basic project structure that we created in the *Installing Grunt on a project* recipe in Chapter 1, *Getting Started with Grunt*. Be sure to refer to it if you are not yet familiar with its contents.

How to do it...

The following steps will take us through creating a simple Stylus style sheet and compiling it to CSS:

1. We'll start by installing the package that contains the `contrib-stylus` plugin, as per the instructions provided in the *Installing a plugin* recipe in Chapter 1, *Getting Started with Grunt*.
2. Let's create a simple Stylus file called `styles.styl` in our project directory with the following contents:

   ```
   first = #ffffff
   second = #000000

   body
     background-color: first
     background-image: url('background.png')
     color: second
   ```

3. Now, we can add the following `stylus` task to our configuration, which will compile the `styles.styl` file to the `styles.css` file in our project directory:

   ```
   stylus: {
     styles: {
       src: 'styles.styl',
       dest: 'styles.css'
     }
   }
   ```

4. Then, we can run the task by using the `grunt stylus` command, which should produce output similar to the following:

   ```
   Running "stylus:styles" (stylus) task
   File styles.css created.
   ```

5. If we now take a look at our project directory, we should see the new `styles.css` file with the following contents:

   ```
   body {

     background-color: #fff;
     background-image: url("background.png");
     color: #000;
   }
   ```

   ### Tip

Note that the contents of this file will probably be in a compressed state due to the `stylus` task compressing its output by default. The contents are presented uncompressed here, for the sake of readability.

There's more...

The `stylus` task provides us with several useful options that can be used in conjunction with its basic compilation feature. We'll look at how to disable compression of the resulting CSS, specify extra import paths, define global variables, enable the inclusion of imported CSS files, and add a banner to the resulting output.

Disabling compression of the resulting CSS

By default, the `stylus` task will compress the resulting CSS by removing all white spaces. This behavior can be disabled by setting the `compress` option to `false`, as per the following example:

```
stylus: {
  styles: {
    options: {
      compress: false
    },
    src: 'styles.styl',
    dest: 'styles.css'
  }
}
```

Specifying extra import paths

In case we'd like to allow our Stylus style sheets to import style sheets other than the ones available to them via a simple relative path, we can indicate extra directories that should be scanned when importing files by using the `paths` option. The following example indicates that the `lib` directory in our project root should be included when scanning for imports:

```
stylus: {
  styles: {
    options: {
      paths: ['lib']
    },
    src: 'styles.styl',
    dest: 'styles.css'
  }
}
```

Defining global variables

By making use of the `define` option, we can define variables that should be made available to all the LESS style sheets that we target. The following example defines the `keyColor` variable that is to be available globally:

```
stylus: {
  styles: {
    options: {
      define: {
        keyColor: '#000000'
      }
    },
    src: 'styles.styl',
    dest: 'styles.css'
  }
}
```

Enabling the inclusion of imported CSS files

When using the `@import` statement in Stylus style sheets to import regular CSS files, the default behavior of the compiler is to produce a standard CSS `@import` directive that references the file. By setting the `'include css'` option to `true`, as per the following example, we can direct the compiler to actually include the specified files into the final result:

```
stylus: {
  styles: {
    options: {
      'include css': true
    },
    src: 'styles.styl',
    dest: 'styles.css'
  }
}
```

Adding a banner to the resulting output

In case we'd like to add some extra information to the beginning of our compiled style sheet, we can make use of the `banner` option. The following example adds a banner containing information about the file we're producing:

```
stylus: {
  styles: {
    options: {
      banner: '/* Name: style.css */\n' +
        '/* Author: Grunt Wiz */\n' +
        '/* License: MIT */\n\n'
    },
    src: 'styles.styl',
    dest: 'styles.css'
  }
}
```

Compiling CoffeeScript to JavaScript

In this recipe, we'll make use of the `contrib-coffee (0.11.0)` plugin to compile **CoffeeScript** source files to **JavaScript**.

Getting ready

In this example, we'll work with the basic project structure we created in the *Installing Grunt on a project* recipe in Chapter 1, *Getting Started with Grunt*. Be sure to refer to it if you are not yet familiar with its contents.

How to do it...

The following steps will take us through creating a simple CoffeeScript source file and compiling it to JavaScript:

1. We'll start by installing the package that contains the `contrib-coffee` plugin, as per the instructions provided in the *Installing a plugin* recipe in Chapter 1, *Getting Started with Grunt*.
2. Let's create a simple CoffeeScript file called `main.coffee` in our project directory with the following contents:

   ```
   logic = (message) ->
     console.log(message)

   message = 'Functionality!'

   logic(message)
   ```

3. Now, we can add the following `coffee` task to our configuration, which will compile the `main.coffee` file to the `main.js` file in our project directory:

   ```
   coffee: {
     main: {
       src: 'main.coffee',
       dest: 'main.js'
     }
   }
   ```

4. Then, we can run the task by using the `grunt coffee` command, which should produce output similar to the following:

   ```
   Running "coffee:main" (coffee) task
   >> 1 files created.
   ```

5. If we now take a look at our project directory, we should see the new `main.js` file with the following contents:

   ```
   (function() {
     var logic, message;

     logic = function(message) {
       return console.log(message);
     };

     message = 'Functionality!';
   ```

```
  logic(message);
}).call(this);
```

There's more...

The `coffee` task provides us with several useful options that can be used in conjunction with its basic compilation feature. We'll look at how to compile without the top-level safety wrapper and concatenate multiple targets before compilation.

Compiling without the top-level safety wrapper

By default, the CoffeeScript compiler wraps the compiled code it produces in an anonymous JavaScript function. This practice prevents the produced code from polluting the global namespace, which can cause a host of problems. This behavior can be disabled by making use of the `bare` option as per the following example:

```
coffee: {
  main: {
    options: {
      bare: true
    },
    src: 'main.coffee',
    dest: 'main.js'
  }
}
```

Concatenating multiple targets before compilation

When specifying that multiple targets should be compiled into one resulting file, the default behavior is to join these files together after they were each compiled individually. With the top-level safety wrapper enabled, this would result in a file that contains the code of each compiled file, wrapped inside its own top-level function.

To join the files before compilation and have all the compiled contents in one top-level wrapper function, we can make use of the `join` option, as per the following example:

```
coffee: {
  main: {
    options: {
      join: true
    },
    src: ['main.coffee', 'more.coffee'],
    dest: 'main.js'
  }
}
```

Compiling LiveScript to JavaScript

In this recipe, we'll make use of the `livescript (0.5.1)` plugin to compile **LiveScript** source files to **JavaScript**.

Getting ready

In this example, we'll work with the basic project structure we created in the *Installing Grunt on a project* recipe in Chapter 1, *Getting Started with Grunt*. Be sure to refer to it if you are not yet familiar with its contents.

How to do it...

The following steps will take us through creating a simple LiveScript source file and compiling it to JavaScript:

1. We'll start by installing the package that contains the `contrib-livescript` plugin, as per the instructions provided in the *Installing a plugin* recipe in Chapter 1, *Getting Started with Grunt*.
2. Let's create a simple LiveScript file called `main.ls` in our project directory with the following contents:

   ```
   logic = (msg) -> console.log(msg)
   message = 'Functionality!'
   logic message
   ```

3. Now, we can add the following `livescript` task to our configuration, which will compile the `main.ls` file to the `main.js` file in our project directory:

   ```
   livescript: {
     main: {
       src: 'main.ls',
       dest: 'main.js'
     }
   }
   ```

4. Then, we can run the task by using the `grunt livescript` command, which should produce output similar to the following:

   ```
   Running "livescript:main" (livescript) task
   File main.js created.
   ```

5. If we now take a look at our project directory, we should see the new `main.js` file with the following contents:

   ```
   (function(){
     var logic, message;
     logic = function(msg){
       return console.log(msg);
     };
     message = 'Functionality!';
     logic(message);
   }).call(this);
   ```

Generating source maps for LESS

In this recipe, we'll make use of the `contrib-less (0.11.4)` plugin to generate **source maps** when compiling our **LESS** style sheets to **CSS**.

Getting ready

In this example, we'll work with the basic project structure that we created in the *Compiling LESS to CSS* recipe of this chapter. Be sure to refer to it if you are not yet familiar with its contents.

How to do it...

The following steps will take us through compiling our LESS style sheet to a CSS file, which contains both our source map and the source style sheet:

1. First, we'll indicate that we'd like to generate a source map by setting the `sourceMap` option to `true` in our `less` task's configuration:

   ```
   less: {
     styles: {
       options: {
         sourceMap: true
       },
       src: 'styles.less',
       dest: 'styles.css'
     }
   }
   ```

2. Next, we'll indicate that we'd like to include the source of the generated CSS in the resulting file by setting the `outputSourceFiles` option to `true` in task's configuration:

   ```
   less: {
     styles: {
       options: {
         sourceMap: true,
         outputSourceFiles: true
       },
       src: 'styles.less',
       dest: 'styles.css'
     }
   }
   ```

3. Then, we can run the task by using the `grunt less` command, which should produce output similar to the following:

   ```
   Running "less:styles" (less) task
   File styles.css created: 99 B → 296 B
   ```

4. If we now take a look at the generated `styles.css` file, we should see the source map and source style sheet embedded near the end of it.

There's more...

As seen in our main recipe, the default behavior for the `less` task is to embed the source map in the generated CSS file. This is the simplest way to both generate and consume a source map, but this method increases the size of the resulting file for all users, which in turn increases the loading time of the web application.

Most developers prefer to keep the source map and the CSS result separate, so that the average user shouldn't have to download the source map and the source file it references. This kind of setup is, however, a little bit more complex due to the fact that it requires the source maps and source files to be made available to the browser, which will make use of the CSS files produced.

The following steps take us through altering the configuration to indicate that the source map and source file should not be included in the resulting CSS file.

1. First, we'll indicate that we'd like to generate a source map by setting the `sourceMap` option to `true` in our `less` task's configuration:

   ```
   less: {
     styles: {
       options: {
         sourceMap: true
       },
       src: 'styles.less',
       dest: 'styles.css'
     }
   }
   ```

2. Then, we'll alter the task to indicate that an external source map should be generated. This is done by specifying a name for the source map file by using the `sourceMapFilename` option:

   ```
   less: {
     styles: {
       options: {
         sourceMap: true,
         sourceMapFilename: 'styles.css.map'
       },
       src: 'styles.less',
       dest: 'styles.css'
     }
   }
   ```

3. Then, we can run the task by using the `grunt less` command, which should produce output similar to the following:

   ```
   Running "less:styles" (less) task
   File styles.css.map created.
   File styles.css created: 99 B → 137 B
   ```

4. If we now take a look at our project directory, we should see the `styles.css` and `styles.css.map` files. The generated style sheet should now also contain a reference to the source map file, and now the source map should also reference the `styles.less` source file.

How it works...

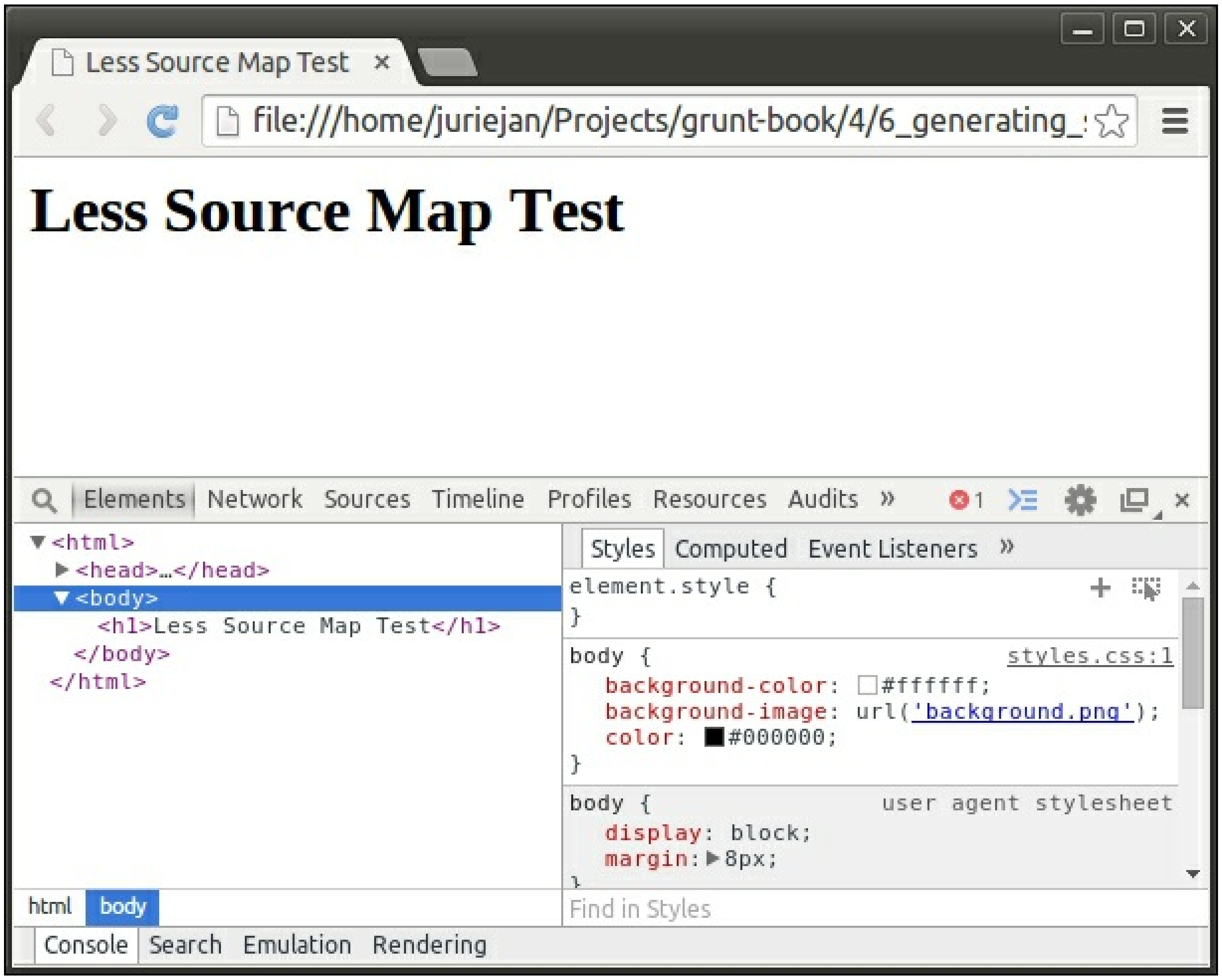
Making use of the generated source map is as simple as including the generated CSS file in a sample HTML file. We would, however, have to ensure that all the necessary files are accessible to the browser that opens the HTML file, which is taken care of in our current example due to all the files being located in the same directory. We must also ensure that the browser we use to open the HTML file provides support for the use of source maps.

The following steps take us through creating a sample HTML file, which includes the generated CSS file, and opening it in a browser that supports source maps.

1. Let's create a sample HTML file, which will include the generated JavaScript source file. Let's create a file in the project directory called `test.html` with the following contents:

   ```
   <html>
     <head>
       <title>Less Source Map Test</title>
       <link rel="stylesheet" type="text/css" href="styles.css">
     </head>
     <body>
       <h1>Less Source Map Test</h1>
     </body>
   </html>
   ```

2. With our sample HTML file ready, we can open it using a browser that supports source maps. For our example, we'll use Google Chrome Version 36. The following image shows the browser window with the developer's tools open and the `body` element selected for inspection.



```
Less Source Map Test
```

```
▼<html>
  ▶<head>…</head>
  ▼<body>
     <h1>Less Source Map Test</h1>
   </body>
 </html>
```

```
element.style {
}
body {                                    styles.css:1
   background-color: #ffffff;
   background-image: url('background.png');
   color: #000000;
}
body {                           user agent stylesheet
   display: block;
   margin: ▶8px;
}
```

3. If we take a look at the styles inspector that is located in the developer's tools, we'll see that `styles.less:4` is being displayed as the location from which the body elements style is being derived. This indicates that our source map is working correctly.

Generating source maps for Sass

In this recipe, we'll make use of the `sass (0.12.1)` plugin to generate **source maps** when compiling our **Sass** style sheets to **CSS**.

Getting ready

In this example, we'll work with the basic project structure that we created in the *Compiling Sass to CSS* recipe in this chapter. Be sure to refer to it if you are not yet familiar with its contents.

How to do it...

The following steps will take us through altering our configuration so that a source map is generated when our Sass style sheet is compiled to CSS:

1. First, we'll indicate that we'd like to generate a source map by setting the `sourceMap` option to `true` in our `sass` task's configuration:

   ```
   sass: {
     styles: {
       options: {
         sourceMap: true
       },
       src: 'styles.scss',
       dest: 'styles.css'
     }
   }
   ```

2. Then, we can run the task by using the `grunt sass` command, which should produce output similar to the following:

   ```
   Running "sass:styles" (sass) task
   File styles.css created.
   File styles.css.map created.
   ```

3. If we now take a look at our project directory, we should see the `styles.css` and `styles.css.map` files. The generated style sheet should now also contain a reference to the source map file, and the source map should now also reference the `styles.scss` source file.

How it works...

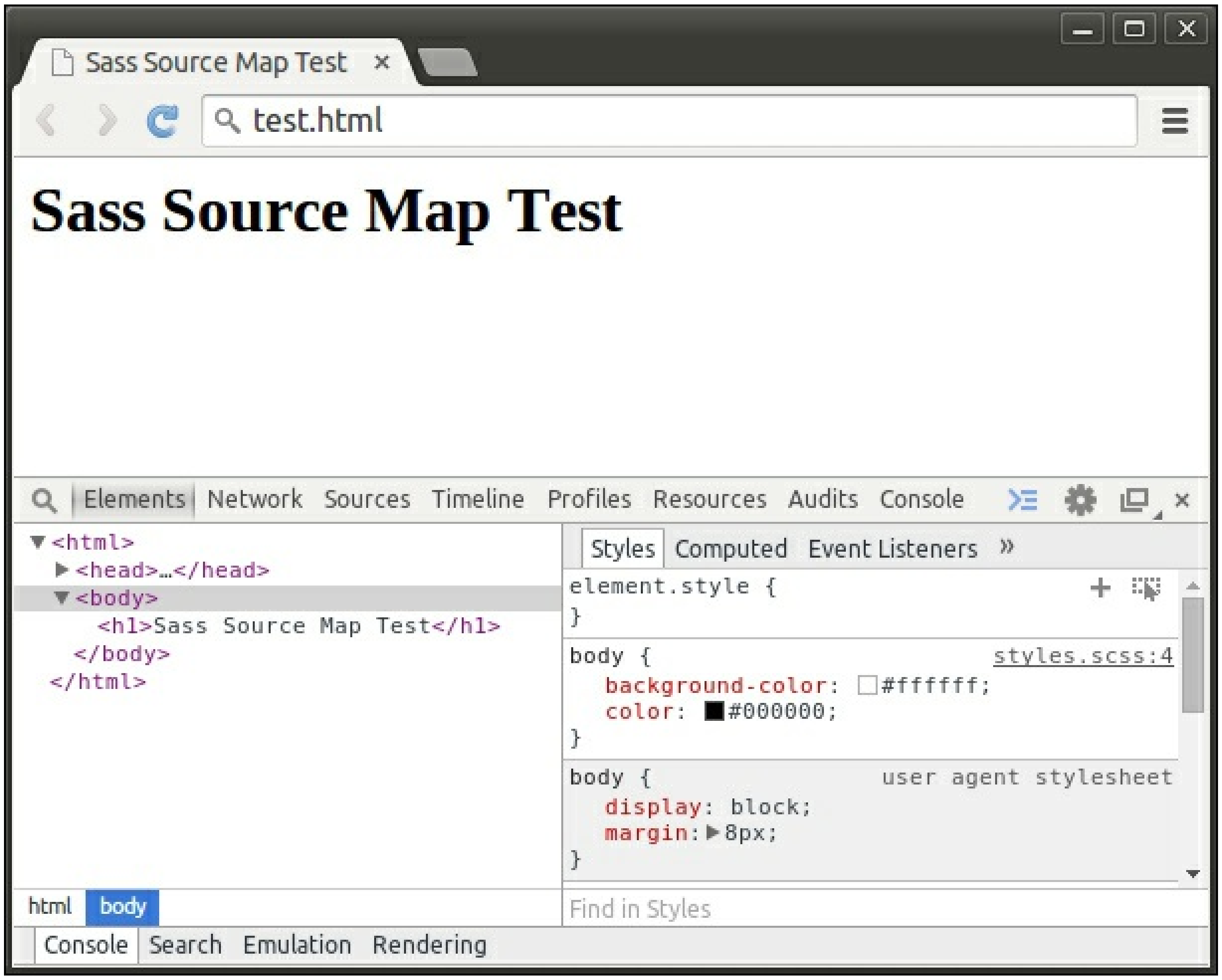
Making use of the generated source map is as simple as including the generated CSS file in a sample HTML file. We will, however, have to ensure that all the necessary files are accessible to the browser that opens the HTML file, which is taken care of in our current example due to all the files being located in the same directory. We must also ensure that the browser we use to open the HTML file provides support for the use of source maps.

The following steps will take us through creating a sample HTML file that includes the generated CSS file and opening it in a browser that supports source maps:

1. Let's create a sample HTML file called `test.html`, which will include the generated JavaScript source file, and provide it with the following contents:

   ```
   <html>
     <head>
       <title>Sass Source Map Test</title>
       <link rel="stylesheet" type="text/css" href="styles.css">
     </head>
     <body>
       <h1>Sass Source Map Test</h1>
     </body>
   </html>
   ```

2. With our sample HTML file ready, we can open it using a browser that supports source maps. For our example, we'll use Google Chrome Version 36. The following image shows the browser window with the developers tools open and the `body` element selected for inspection.


Sass Source Map Test
test.html
Elements Network Sources Timeline Profiles Resources Audits Console
Styles Computed Event Listeners
Find in Styles
html body
Console Search Emulation Rendering

Sass Source Map Test

```
<html>
  <head>…</head>
  <body>
    <h1>Sass Source Map Test</h1>
  </body>
</html>
```

```
element.style {
}
body {                                  styles.scss:4
   background-color: #ffffff;
   color: #000000;
}
body {                          user agent stylesheet
   display: block;
   margin: 8px;
}
```

3. If we take a look at the styles inspector that is located in the developer's tools, we'll see that `styles.scss:4` is being displayed as the location from where the body elements style is being derived. This indicates that our source map is working correctly.

Generating source maps for CoffeeScript

In this recipe, we'll make use of the `contrib-coffee (0.11.0)` plugin to generate **source maps** when compiling our **CoffeeScript** source files to **JavaScript**.

Getting ready

In this example, we'll work with the basic project structure that we created in the *Compiling CoffeeScript to JavaScript* recipe of this chapter. Be sure to refer to it if you are not yet familiar with its contents.

How to do it...

The following steps will take us through altering our configuration to generate a source map when compiling our CoffeeScript source files to JavaScript:

1. First, we'll indicate that we'd like to generate a source map by setting the `sourceMap` option to `true` in our `coffee` task's configuration:

   ```
   sass: {
     styles: {
       options: {
         sourceMap: true
       },
       src: 'styles.scss',
       dest: 'styles.css'
     }
   }
   ```

2. Then, we can run the task by using the `grunt coffee` command, which should produce output similar to the following:

   ```
   Running "coffee:main" (coffee) task
   >> 1 files created.
   >> 1 source map files created.
   ```

3. If we now take a look at our project directory, we should see the `main.js` and `main.js.map` files. The generated JavaScript source file should now also contain a reference to the source map file, and the source map should now also reference the `main.coffee` source file.

How it works...

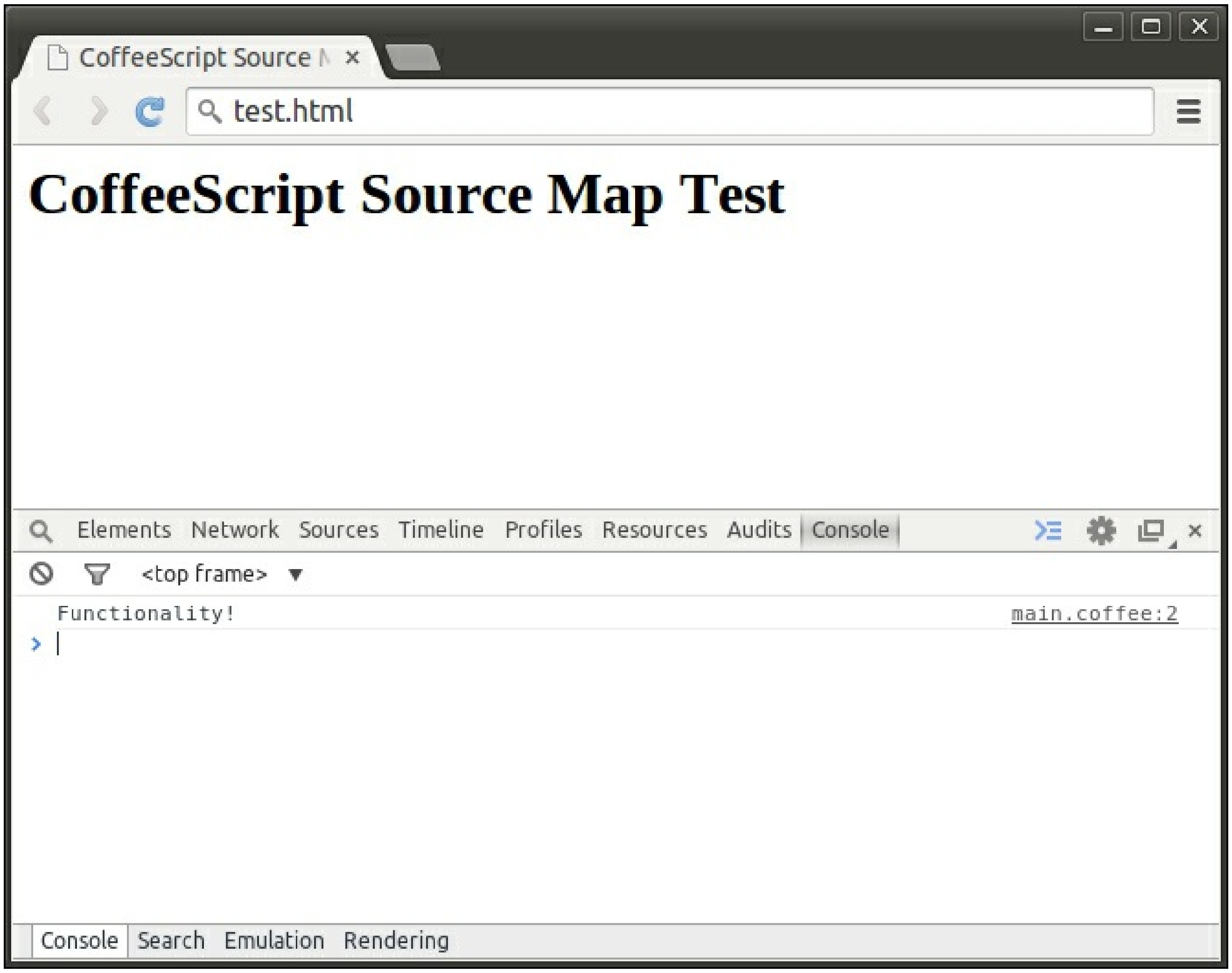
Making use of the generated source map is as simple as including the generated CSS file in a sample HTML file. We will, however, have to ensure that all the necessary files are accessible to the browser that opens the HTML file, which is taken care of in our current example due to all the files being located in the same directory. We must also ensure that the browser we use to open the HTML file provides support for the use of source maps.

The following steps take us through creating a sample HTML file that includes the generated JavaScript file and opening the HTML file in a browser that supports source maps:

1. Let's create a sample HTML file called `test.html`, which will include the generated JavaScript source file, and provide it with the following contents:

   ```
   <html>
     <head>
       <title>CoffeeScript Source Map Test</title>
       <script type="text/javascript" src="main.js"></script>
     </head>
     <body>
       <h1>CoffeeScript Source Map Test</h1>
     </body>
   </html>
   ```

2. With our sample HTML file ready, we can open it using a browser that supports source maps. For our example, we'll use Google Chrome Version 36. The following image shows the browser window with the developer's tools open:



CoffeeScript Source Map Test

```
Functionality!                                              main.coffee:2
```

Tip

You may have to refresh the browser after opening the developer's tools in order to see the appropriate console output discussed in the next step.

3. If we take a look at the output in the console located in the developer's tools, we'll see that `main.coffee:2` is being displayed as the location from where the `'Functionality!'` string is being logged. This indicates that our source map is working correctly.

Defining custom functions with LESS

In this recipe, we'll make use of the `contrib-less (0.11.4)` plugin to define custom functions that can be used in our **LESS** style sheets.

For our example, we'll create a custom function called `halfDarken`, which will darken any color provided to it by 50 percent. It will make use of the LESS library's built-in `darken` function.

Getting ready

In this example, we'll work with the basic project structure that we created in the *Compiling LESS to CSS* recipe of this chapter. Be sure to refer to it if you are not yet familiar with its contents.

How to do it...

The following steps will take us through altering our configuration to define a custom function, and altering our LESS style sheet to use it:

1. First, we'll alter the configuration of our `less` task by adding the `customFunctions` option and defining a stub for the `halfDarker` function:

   ```
   less: {
     styles: {
       options: {
         customFunctions: {
           halfDarken: function (less, color) {
             return color;
           }
         }
       },
       src: 'styles.less',
       dest: 'styles.css'
     }
   }
   ```

2. Then, we can add some logic for the `halfDarken` function that darkens the provided color by 50 percent and returns the result:

   ```
   halfDarken: function (less, color) {
     functions = less.tree.functions;
     return functions.darken(color, {value:50});
   }
   ```

3. Once we've got the function defined, we can alter our `styles.less` file to make use of it:

   ```
   @first: #ffffff;
   @second: #000000;

   body {
     background-color: halfDarken(@first);
     color: @second;
   }
   ```

4. We can now run our task by using the `grunt less` command, which should produce the following output:

   ```
   Running "less:styles" (less) task
   File styles.css created: 0 B → 56 B
   ```

5. If we now take a look inside the resulting `styles.css` file, we can see that the `halfDarken` function has made the color that was provided to it 50 percent darker:

   ```
   body {
     background-color: #808080;
     color: #000000;
   }
   ```

Using Stylus plugins

In this recipe, we'll make use of the `contrib-stylus (0.18.0)` plugin to compile **Stylus** style sheets that make use of one of the Stylus libraries provided by the developer community.

Tip

A partial list of the plugins available for the Stylus framework can be found at the following URL:

https://github.com/LearnBoost/stylus/wiki

For our example, we'll make use of the **axis** library that comes packaged with the **Roots** platform. It provides a host of useful functions, mixins, and other utilities that you'll be sure to make use of in your day-to-day style sheet development.

Getting ready

In this example, we'll work with the basic project structure that we created in the *Compiling Stylus to CSS* recipe of this chapter. Be sure to refer to it if you are not yet familiar with its contents.

How to do it...

The following steps will take us through altering our configuration in order to provide our style sheets with access to the axis library:

1. First, we'll need to install the `axis-css` package on our local project path, using the `npm install --save axis-css` command. This should produce output that contains the following:

   ```
   axis-css@0.1.8 node_modules/axis-css
   ```

2. With the library now installed, we can alter the configuration of the `stylus` task in such a way that the library is loaded before the target style sheets are compiled. This is done with the `use` option that accepts an array of imported libraries that are prepared to be used as Stylus plugins. The following configuration uses this option to load the `axis-css` plugin:

   ```
   stylus: {
     styles: {
       options: {
         use: [
           require('axis-css')
         ]
       },
       src: 'styles.styl',
       dest: 'styles.css'
     }
   }
   ```

3. As a test, we can now also make use of a feature provided by the axis library in our sample style sheet. Let's alter the contents of `main.styl` to include the special `absolute` property provided by the library:

   ```
   first = #ffffff
   second = #000000

   body
     absolute: top left
     background-color: first
     color: second
   ```

4. We can now test our setup by running the `grunt stylus` command, which should produce output similar to the following:

   ```
   Running "stylus:styles" (stylus) task
   File styles.css created.
   ```

5. If we take a look inside the generated `main.css` file, we'll now see that it contains positional properties that were generated by the axis library's `absolute` property:

```
body {
  position: absolute;
  top: 0;
  left: 0;
  background-color: #fff;
  color: #000;
}
```

Tip

Note that the contents of this file will probably be in a compressed state due to the stylus task compressing its output by default. The contents are presented uncompressed here for the sake of readability.

Chapter 5. Running Automated Tests

In this chapter, we will cover the following recipes:

- Running Jasmine tests
- Running QUnit tests
- Running NodeUnit tests
- Running Mocha client-side tests
- Running Mocha server-side tests
- Generating coverage reports for server-side code using Mocha and Blanket
- Generating coverage reports for client-side code using Mocha and Blanket
- Generating coverage reports for client-side code using QUnit and Istanbul

Introduction

As the size and complexity of a software unit increases, it can become quite time-consuming to ensure that it behaves according to its specifications each time it is altered. For this purpose, automated testing becomes invaluable by increasing the overall reliability and quality of a software unit, without constant manual testing.

There are various levels of testing that a project can implement, ranging from unit tests at the function or class level, up to integration tests that make use of an entire application stack. Most testing frameworks provide for this entire range, perhaps just with the addition of a few tools.

Also worth mentioning in relation to testing is the practice of test-driven development, in which a developer first creates (initially failing) a test case for a desired improvement or a new feature, and then does the minimum amount of development to make the test case pass. To finish it off, the developer will then review the written code and refactor it to acceptable standards.

Running Jasmine tests

In this recipe, we'll make use of the `contrib-jasmine (0.7.0)` plugin to run our automated **Jasmine** tests in a **PhantomJS** browser environment.

Getting ready

In this example, we'll work with the basic project structure we created in the *Installing Grunt on a project* recipe in Chapter 1, *Getting Started with Grunt*. Be sure to refer to it if you are not yet familiar with its contents.

How to do it...

The following steps take us through creating a sample code base, a few tests to run against the code base, and setting up Grunt to run them for us in a PhantomJS browser environment.

1. We'll start by installing the package that contains the `contrib-jasmine` plugin by following the instructions provided in the *Installing a plugin* recipe in Chapter 1, *Getting Started with Grunt*.
2. Then, we'll create a simple JavaScript source file in our project directory, which contains a function that we'd like to test. Let's call it `main.js` and add the following content to it:

   ```
   function square(x) {
     return x * x;
   }
   ```

3. Now, we can write a simple suite of tests, using the Jasmine framework, to test the `square` method. Let's create a file called `tests.js` in the project directory with the following contents:

   ```
   describe('Square method', function() {
     it('returns 4 for 2', function () {
       expect(square(2)).toBe(4);
     });
     it('returns 9 for 3', function () {
       expect(square(3)).toBe(9);
     });
     it('returns 16 for 4', function () {
       expect(square(4)).toBe(16);
     });
   });
   ```

4. With our code base and tests created, we can now add the following `jasmine` task to our configuration, which will load the code from `main.js`, and run the tests in the `tests.js` file:

   ```
   jasmine: {
     src: 'main.js',
     options: {
       specs: 'tests.js'
     }
   }
   ```

5. We can then run the task using the `grunt jasmine` command, which should

produce the following output informing us of the test results:

```
Running "jasmine:src" (jasmine) task
Testing jasmine specs via PhantomJS

 Square method
   ✓ returns 4 for 2
   ✓ returns 9 for 3
   ✓ returns 16 for 4

3 specs in 0.015s.
>> 0 failures
```

There's more...

The `jasmine` task provides us with several useful options that can be used in conjunction with its basic test running feature. We'll look at how to load helpers to be used in tests, how to load libraries before running tests, how to load styles required by tests, and how to provide a custom template for the specification runner.

Loading helpers to be used in tests

If we'd like to make use of custom equality testers or matchers, we can include them using the `helpers` option before tests are run. In the following example, we indicate that the custom helpers contained in the `helpers.js` file should be loaded before running the tests:

```
jasmine: {
  src: 'main.js',
  options: {
    specs: 'tests.js',
    helpers: 'helpers.js'
  }
}
```

Loading libraries before running tests

In case the code we'd like to test depends on third-party libraries that you don't load in either your source, specifications or helpers, they can be loaded using the `vendor` option as shown in the following example:

```
jasmine: {
  src: 'main.js',
  options: {
    specs: 'tests.js',
    vendor: ['lodash.min.js']
  }
}
```

Loading styles required by tests

If we have tests that depend on specific CSS styles being present in the browser, we can have them loaded using the `styles` option as shown in the following example:

```
jasmine: {
  src: 'main.js',
  options: {
    specs: 'tests.js',
    styles: 'styles.css'
  }
}
```

Providing a custom template for the specification runner

When writing tests for source code that runs inside a browser, the need for a few HTML elements, such as fixtures, can be quite common. The simplest way to add HTML to the generated specification runner (`test.html`) is to customize the template that it's generated with.

The following steps take us through retrieving the default specification runner template, customizing it, and using it as our template:

1. The default specification runner template can be retrieved from the repository of the `contrib-jasmine` plugin and saved to the `runner.tmpl` file.

   ### Tip

   At the time of writing this, the default specification runner template could be downloaded from the following link:

   https://github.com/gruntjs/grunt-contrib-jasmine/raw/master/tasks/jasmine/templates/DefaultRunner.tmpl

2. Once we have the default template saved as `runner.tmpl`, we can make some alterations to it. In the following example, we'll just add an element with some text:

   ```
   <!doctype html>
   <html>
   <head>
     <meta charset="utf-8">
     <title>Jasmine Spec Runner</title>
     <link rel="shortcut icon" type="image/png" href="<%= temp
   %>/jasmine_favicon.png">
     <% css.forEach(function(style){ %>
       <link rel="stylesheet" type="text/css" href="<%= style %>">
     <% }) %>
   </head>
   <body>
     <% with (scripts) { %>
       <% [].concat(polyfills, jasmine, boot, vendor, helpers, src,
   specs,reporters).forEach(function(script){ %>
       <script src="<%= script %>"></script>
       <% }) %>
   ```

```
  <% }; %>
  <div id="test">Test</div>
</body>
</html>
```

3. With the custom template ready, we'll make use of the `template` option to indicate that it should be used in the generation of the runner:

```
jasmine: {
  src: 'main.js',
  options: {
    specs: 'tests.js',
    template: 'runner.tmpl'
  }
}
```

4. This will now make the `test` element available to us in our tests, allowing us to include tests similar to the following in our specifications:

```
describe('Test element', function() {
  it('has test text', function () {
    expect(window.test.innerHTML).toBe("Test");
  });
});
```

Running QUnit tests

In this recipe, we'll make use of the `contrib-qunit (0.5.2)` plugin to run our automated **QUnit** tests in a **PhantomJS** browser environment.

Getting ready

In this example, we'll work with the basic project structure we created in the *Installing Grunt on a project* recipe in Chapter 1, *Getting Started with Grunt*. Be sure to refer to it if you are not yet familiar with its contents.

How to do it...

The following steps take us through creating a sample code base, a few tests to run against the code base, setting up a testing environment, and configuring Grunt to run them for us in a PhantomJS browser.

1. We'll start by installing the package that contains the `contrib-qunit` plugin by following the instructions provided in the *Installing a plugin* recipe in Chapter 1, *Getting Started with Grunt*.
2. Then, we'll create a simple JavaScript source file in our project directory, which contains a function that we'd like to test. Let's call it `main.js` and provide it with the following contents:

   ```
   function square(x) {
     return x * x;
   }
   ```

3. Now, we can write a simple set of tests, using the QUnit framework, for the `square` method. Let's create a file called `tests.js` in the project directory with the following contents:

   ```
   QUnit.test("Square method functionality", function(assert) {
     assert.equal(square(2), 4);
     assert.equal(square(3), 9);
     assert.equal(square(4), 16);
   });
   ```

4. Due to the tests being run inside a browser and the `contrib-qunit` plugin not automatically including it, we'll have to download the QUnit library and style sheet into the project directory.

   ### Tip

   At the time of writing this, the QUnit library and its accompanying style sheet could be downloaded from the following links:

   http://code.jquery.com/qunit/qunit-1.15.0.js

   http://code.jquery.com/qunit/qunit-1.15.0.css

5. To bring together all the parts we set up in the previous steps, we now need to create a testing environment that loads our code base, tests, and all the required libraries. Let's create the `test.html` file in our project directory with the

following contents:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>QUnit basic example</title>
    <link rel="stylesheet" href="qunit-1.15.0.css">
    <script src="qunit-1.15.0.js"></script>
    <script src="main.js"></script>
    <script src="tests.js"></script>
  </head>
  <body>
    <div id="qunit"></div>
    <div id="qunit-fixture"></div>
  </body>
</html>
```

6. With the code base, tests, and testing environment now in place, we can add the following `qunit` task to our configuration:

```
qunit: {
  main: {
    src: 'test.html'
  }
}
```

7. We can then run the task using the `grunt qunit` command, which should produce the following output informing us of the test results:

```
Running "qunit:main" (qunit) task
Testing test.html .OK
>> 3 assertions passed (18ms)
```

There's more...

The `qunit` task provides us with several useful options that can be used in conjunction with its basic test running feature. We'll look at loading tests from a web server, continuing execution after failed tests, suppressing the PhantomJS browser console output, and passing arguments to the PhantomJS instance.

Loading tests from a web server

If we'd like to load our testing environment along with all its parts from a web server instead of straight from a file on the filesystem, we can make use of the `urls` option by providing it with the absolute URLs of the testing environments that we'd like to run.

The following example takes us through moving the required files to a directory they can be served from, setting up a basic web server that serves them, and altering our configuration to test the files from the web server.

1. We'll start by installing the package that contains the `contrib-connect` plugin by following the instructions provided in the *Installing a plugin* recipe in Chapter 1, *Getting Started with Grunt*.
2. Then, we can create a directory called `www` in the project root and move the `main.js`, `qunit-1.15.0.css`, `qunit-1.15.0.js`, `test.html`, and `tests.js` files into it.
3. Now, we can set up the basic web server that will serve the files from the `www` directory by adding the following `connect` task configuration:

```
connect: {
  server: {
    options: {
      base: 'www'
    }
  }
}
```

4. With the server ready to serve the files required by our testing environment, we can alter the configuration of our `qunit` task to load it from the appropriate URL:

```
qunit: {
  main: {
    options: {
      urls: ['http://localhost:8000/test.html']
    }
  }
```

```
    }
```

5. To test our setup, we can run the `grunt connect qunit` command to start the web server and run the testing environment hosted on it. This should produce output similar to the following:

   ```
   Running "connect:server" (connect) task
   Started connect web server on http://0.0.0.0:8000

   Running "qunit:main" (qunit) task
   Testing http://localhost:8000/test.html .OK
   >> 3 assertions passed (19ms)
   ```

Continuing execution after failed tests

The default behavior of the `qunit` task is to fail the entire task if a failure occurs in any one of the tests. This will in turn cause any of the tasks lined up for execution after the `qunit` task not to be executed. By setting the `force` option to `true`, as we do in the following example, we can indicate that the task itself should not fail due to test failures:

```
qunit: {
  main: {
    src: 'test.html',
    options: {
      force: true
    }
  }
}
```

Suppressing the PhantomJS browser console output

The default behavior of the `qunit` task is to print the console output generated in the headless PhantomJS browser to the command line where the task runs. If we'd like to prevent the console output from being printed, we can set the `console` option to `false` as shown in the following example:

```
qunit: {
  main: {
    src: 'test.html',
    options: {
      console: false
    }
  }
}
```

Passing arguments to the PhantomJS instance

In case we'd like to pass some options to the PhantomJS process when it starts, we can provide them along with the regular options just as they will be provided on the command line.

Tip

A list of arguments that the PhantomJS executable accepts can be found at the following URL:

http://phantomjs.org/api/command-line.html

The following example disables the loading of images in the browser by setting the `load-images` option to `false`:

```
qunit: {
  main: {
    src: 'test.html',
    options: {
      '--load-images': false
    }
  }
}
```

Running NodeUnit tests

In this recipe, we'll make use of the `contrib-nodeunit (0.4.1)` plugin to run our automated **NodeUnit** tests.

Getting ready

In this example, we'll work with the basic project structure we created in the *Installing Grunt on a project* recipe in Chapter 1, *Getting Started with Grunt*. Be sure to refer to it if you are not yet familiar with its contents.

How to do it...

The following steps take us through creating a sample code base, creating a few tests to run against the code base, and configuring Grunt to run them for us.

1. We'll start by installing the package that contains the `contrib-nodeunit` plugin by following per the instructions provided in the *Installing a plugin* recipe in Chapter 1, *Getting Started with Grunt*.
2. Then, we'll create a simple JavaScript source file in our project directory that contains and exports a function that we'd like to test. Let's call it `main.js` and add the following content to it:

   ```
   module.exports.square = function (x) {
     return x * x;
   }
   ```

3. Now, we can write a simple set of tests, using the NodeUnit framework, to test the `square` method. We will also have to import the `square` method into our suite of tests, since the `nodeunit` task does not do this automatically. Let's create a file called `tests.js` in the project directory with the following contents:

   ```
   var square = require('./main').square;

   module.exports.testSquare = {
     'Square method returns 4 for 2': function (test) {
       test.equal(square(2), 4);
       test.done();
     },
     'Square method returns 9 for 3': function (test) {
       test.equal(square(2), 4);
       test.done();
     },
     'Square method returns 16 for 4': function (test) {
       test.equal(square(2), 4);
       test.done();
     }
   }
   ```

4. With our code base and tests created, we can now add the following `nodeunit` task to our configuration, which will run the tests contained in the `tests.js` file:

   ```
   nodeunit: {
     main: {
       src: 'tests.js'
     }
   }
   ```

5. We can then run the task using the `grunt nodeunit` command, which will produce the following output informing us of the test results:

```
Running "nodeunit:main" (nodeunit) task
Testing tests.js...OK
>> 3 assertions passed (10ms)
```

There's more...

The `nodeunit` task provides us with several useful options that can be used in conjunction with its basic test-running feature. We'll look at how to use an alternative reporter and send reporter output to a file.

Using an alternative reporter

In case we'd like to alter the way that the test results are displayed, we can make use of an alternative reporter by specifying one using the `reporter` option.

Tip

The default value of the `reporter` option is `grunt`, but it can be set to any one of the reporters listed in the `module.exports` object at the following URL:

https://github.com/caolan/nodeunit/blob/master/lib/reporters/index.js

In the following example, we use the `reporter` option to indicate that we'd like the results of our tests to be reported in the HTML format:

```
nodeunit: {
  main: {
    src: 'tests.js',
    options: {
      reporter: 'html'
    }
  }
}
```

Sending reporter output to a file

If we'd like the reported results of our tests to be stored in a file when the `nodeunit` task is run, we can indicate which file should receive it using the `reporterOutput` option, as shown in the following example:

```
nodeunit: {
  main: {
    src: 'tests.js',
    options: {
      reporterOutput: 'result.txt'
    }
  }
}
```

Running Mocha client-side tests

In this recipe, we'll make use of the `mocha (0.4.11)` plugin to run our automated **Mocha** tests in a **PhantomJS** browser environment.

Getting ready

In this example, we'll work with the basic project structure we created in the *Installing Grunt on a project* recipe in Chapter 1, *Getting Started with Grunt*. Be sure to refer to it if you are not yet familiar with its contents.

How to do it...

The following steps take us through creating a sample code base, creating a few tests to run against the code base, and configuring Grunt to run them for us in a PhantomJS browser environment.

1. We'll start by installing the package that contains the `mocha` plugin by following the instructions provided in the *Installing a plugin* recipe in Chapter 1, *Getting Started with Grunt*.
2. Then, we'll create a simple JavaScript source file in our project directory that contains a function that we'd like to test. Let's call it `main.js` and provide it with the following contents:

   ```
   function square(x) {
     return x * x;
   }
   ```

3. Now, we can write a simple suite of tests, using the Mocha and **Expect.js** frameworks, for the `square` method. Let's create a file called `tests.js` in the project directory with the following contents:

   ```
   describe('Square method', function() {
     it('returns 4 for 2', function () {
       expect(square(2)).to.be(4);
     });
     it('returns 9 for 3', function () {
       expect(square(3)).to.be(9);
     });
     it('returns 16 for 4', function () {
       expect(square(4)).to.be(16);
     });
   });
   ```

4. Due to the tests being run inside a browser and the `mocha` plugin not automatically including it, we'll have to download the Mocha library and style sheet into the project directory.

   ### Tip

   At the time of writing this, the Mocha library and its accompanying style sheet could be downloaded from the following links:

   https://github.com/visionmedia/mocha/raw/master/mocha.js https://github.com/visionmedia/mocha/raw/master/mocha.css

5. Seeing as the Mocha framework does not include the Expect.js assertion library by default, we also need to download it into our project directory.

   Tip

   At the time of writing this, the Expect.js library could be downloaded from the following link:

   https://github.com/LearnBoost/expect.js/raw/master/index.js

   Note that for our current example, we'll download the `index.js` file mentioned, and change its filename to `expect.js`.

6. To bring together all the parts we set up, we now need to create a testing environment that loads our code base, tests, and all the required libraries. Let's create the `test.html` file in our project directory with the following contents:

```
<html>
  <head>
    <meta charset="utf-8">
    <title>Mocha Tests</title>
    <link rel="stylesheet" href="mocha.css" />
  </head>
  <body>
    <div id="mocha"></div>
    <script src="main.js"></script>
    <script src="expect.js"></script>
    <script src="mocha.js"></script>
    <script>
      mocha.setup('bdd');
    </script>
    <script src="tests.js"></script>
  </body>
</html>
```

7. With our code base, tests, and testing environment ready, we can now add the following `mocha` task to our configuration:

```
mocha: {
  main: {
    src: 'test.html',
    options: {
      run: true
    }
  }
}
```

Tip

The `mocha` task injects code into the PhantomJS browser in order to gather the results of the tests once they have finished running. The `mocha.run` method call required to start the execution of the tests contained in the environment needs to run after the injection of this code in order for the results to be captured. Setting the `run` option to `true` ensures that this method is called after the injection of the code is completed.

8. We can then run the task using the `grunt mocha` command, which should produce the following output informing us of the test results:

```
Running "mocha:main" (mocha) task
Testing: test.html

  3 passing (105ms)
>> 3 passed! (0.10s)
```

There's more...

The `mocha` task provides us with several useful options that can be used in conjunction with its basic test-running feature. We'll look at loading tests from a web server, sending reporter output to a file, displaying the PhantomJS browser's console output, displaying source errors, specifying options for the Mocha test runner, and using an alternative reporter.

Loading tests from a web server

If we'd like to load our testing environment along with all its parts from a web server instead of straight from a file on the filesystem, we can make use of the `urls` option by passing the absolute URLs of the testing environments that we'd like to run.

The following example takes us through moving the required files to a directory they can be served from, setting up a basic web server that serves them, and altering our configuration to test the files from the web server.

1. We'll start by installing the package that contains the `contrib-connect` plugin by following the instructions provided in the *Installing a plugin* recipe in Chapter 1, *Getting Started with Grunt*.
2. Then, we can create a directory called `www` in the project root and move the `expect.js`, `main.js`, `mocha.css`, `mocha.js`, `test.html`, and `tests.js` files into it.
3. Now, we can set up the basic web server, which will serve the files from the `www` directory by adding the following `connect` task configuration:

```
connect: {
  server: {
    options: {
      base: 'www'
    }
  }
}
```

4. With the server ready to serve the files for the testing environment, we can now alter the configuration of our `mocha` task to load it from the appropriate URL:

```
mocha: {
  main: {
    options: {
      run: true,
      urls: ['http://localhost:8000/test.html']
      }
    }
  }
```

5. To test our setup, we can run the `grunt connect qunit` command to start the web server and run the testing environment hosted on it. This should produce output similar to the following:

   ```
   Running "connect:server" (connect) task
   Started connect web server on http://0.0.0.0:8000

   Running "mocha:main" (mocha) task
   Testing: http://localhost:8000/test.html

     3 passing (110ms)
   >> 3 passed! (0.11s)
   ```

Sending reporter output to a file

If we'd like the reported results of our tests to be stored in a file when the `mocha` task is run, we can indicate which file should receive it using the `dest` option as shown in the following example:

```
mocha: {
  main: {
    src: 'test.html',
    dest: 'result.txt',
    options: {
      run: true
    }
  }
}
```

Displaying the PhantomJS browser's console output

By default, the output from the Phantom JS browser's console will not be displayed in the command-line output when the `mocha` task runs. If we'd like to display the output, we can set the `log` option to `true` as shown in the following example:

```
mocha: {
  main: {
    src: 'test.html',
    options: {
      run: true,
      log: true
    }
```

```
  }
}
```

Displaying source errors

The default behavior for the `mocha` task is to ignore syntax errors encountered in the source files that the tests will run against. If an error is encountered, the source will simply not load, and probably cause all the tests to fail without a useful error message.

If we'd like to be informed of the errors encountered in the source, we can set the `logErrors` option to `true` as shown in the following example:

```
mocha: {
  main: {
    src: 'test.html',
    options: {
      run: true,
      logErrors: true
    }
  }
}
```

Specifying options for the Mocha test runner

In case we'd like to specify some options directly to the Mocha test runner that runs behind the scenes of the `mocha` task, we can provide them using the `mocha` option. The following example uses the `grep` option to indicate that only tests that contain the string `'2'` in their description should be run:

```
mocha: {
  main: {
    src: 'test.html',
    options: {
      run: true,
      mocha: {
        grep: '2'
      }
    }
  }
}
```

Tip

A list of the available options along with their explanations can be found at the following URL:

http://mochajs.org/#usage

Using an alternative reporter

In the case that we'd like to alter the way that the test results are displayed, we can make use of an alternative reporter by specifying one using the `reporter` option.

Tip

The default value of the `reporter` option is `dot` but it can be set to any one of the reporters listed at the following URL:

http://mochajs.org/#reporters

Note that the names of the various reporters as they are listed at the mentioned URL will, for the most part, have to be capitalized when indicating them with the `reporter` option.

In the following example, we use the `reporter` option to indicate that we'd like the results of our tests to be reported in the JSON format:

```
mocha: {
  main: {
    src: 'test.html',
    options: {
      run: true,
      reporter: 'JSON'
    }
  }
}
```

Running Mocha server-side tests

In this recipe, we'll make use of the `mocha-test (0.11.0)` plugin to run our automated **Mocha** tests.

Getting ready

In this example, we'll work with the basic project structure we created in the *Installing Grunt on a project* recipe in Chapter 1, *Getting Started with Grunt*. Be sure to refer to it if you are not yet familiar with its contents.

How to do it...

The following steps take us through creating a sample code base, creating a few tests to run against the code base, and configuring Grunt to run them for us.

1. We'll start by installing the package that contains the `mocha-test` plugin by following the instructions provided in the *Installing a plugin* recipe in Chapter 1, *Getting Started with Grunt*.
2. Then, we can create a JavaScript source file in our project directory that contains and exports a function that we'd like to have tested. Let's call it `main.js` and add the following content to it:

```
module.exports.square = function (x) {
  return x * x;
}
```

3. Now, we can write a simple set of tests, using the Mocha framework, to test the `square` method. We would also have to import the `square` method and the `assert` library into our suite of tests, since the `mochaTest` task does not do this automatically. Let's create a file called `tests.js` in the project directory with the following contents:

```
var assert = require('assert');
var square = require('./main').square;

describe('Square method', function() {
  it('returns 4 for 2', function () {
    assert.equal(square(2), 4);
  });
  it('returns 9 for 3', function () {
    assert.equal(square(3), 9);
  });
  it('returns 16 for 4', function () {
    assert.equal(square(4), 16);
  });
});
```

4. With our code base and tests created, we can now add the following `mochaTest` task to our configuration, which will run the tests contained in the `tests.js` file:

```
mochaTest: {
  main: {
    src: 'tests.js'
  }
}
```

5. We can then run the task using the `grunt mochaTest` command, which should produce output informing us of the test results, similar to the following:

```
Running "mochaTest:main" (mochaTest) task

  3 passing (6ms)
```

There's more...

The `mochaTest` task provides us with several useful options that can be used in conjunction with its basic test running feature. We'll look at how to use an alternative reporter, select tests using a regular expression, check for global variable leaks, send reporter output to a file, and load extra modules into the testing environment.

Using an alternative reporter

In the case that we'd like to alter the way that the test results are displayed, we can make use of an alternative reporter by specifying one using the `reporter` option.

Tip

The default value of the `reporter` option is `dot` but it can be set to any one of the reporters listed at the following URL:

http://mochajs.org/#reporters

Note that the names of the various reporters as they are listed at the mentioned URL should all be written in lowercase when referring to them with this plugin.

In the following example, we use the `reporter` option to indicate that we'd like the results of our tests to be reported in the JSON format:

```
mochaTest: {
  main: {
    src: 'tests.js',
    options: {
      reporter: 'json'
    }
  }
}
```

Selecting tests using a regular expression

Quite often, we'd like to test only a subset of the tests available in our testing suite. We can target specific tests by providing a **regular expression** to the `grep` option, which will be matched against the description of the tests in our suite.

In the following example, we indicate that we only want to run tests that have a description that contains the `'2'` string:

```
mochaTest: {
  main: {
    src: 'tests.js',
    options: {
      grep: '2'
    }
  }
}
```

Checking for global variable leaks

It's generally considered bad practice to make use of global variables in JavaScript, as collisions can easily occur between variable names used by various libraries, or implied by the JavaScript environment.

If we'd like to receive warnings whenever global variables are encountered in either the source or test code, we can set the `ignoreLeaks` option to `false`. In addition to that, we can also make use of the `globals` option to indicate the variable names that should be ignored when defined as `globals`.

The following example turns global leak detection on, and also indicates that the `allowedGlobal` variable name should be ignored if it is defined globally.

```
mochaTest: {
  main: {
    src: 'tests.js',
    options: {
      ignoreLeaks: false,
      globals: ['allowedGlobal']
    }
  }
}
```

Sending reporter output to a file

If we'd like the reported results of our tests to be stored in a file when the `mochaTest` task is run, we can indicate which file should receive it using the `captureFile` option as shown in the following example:

```
mochaTest: {
  main: {
    src: 'tests.js',
    options: {
      captureFile: 'result.txt'
    }
  }
}
```

Loading extra modules into the testing environment

The `require` option can be used to load modules into our testing environment before the tests are run. This allows us to make use of libraries without having to import them into each of our test suites.

In the following example, we load the `should.js` module so that we can make use of the **Should.js** library in our tests:

```
mochaTest: {
  main: {
    src: 'tests.js',
    options: {
      require: ['should']
    }
  }
}
```

Tip

Note that for this example to work, the `should` package needs to be installed either locally or globally.

Generating coverage reports for server-side code using Mocha and Blanket

In this recipe, we'll make use of the `blanket (0.0.8)`, `mocha-test (0.11.0)`, and `contrib-copy (0.5.0)` plugins to run our automated **Mocha** tests, while at the same time, generating **coverage reports** for the source code they run against.

Getting ready

In this example, we'll work with the basic project structure we created in the *Installing Grunt on a project* recipe in Chapter 1, *Getting Started with Grunt*. Be sure to refer to it if you are not yet familiar with its contents.

How to do it...

The following steps take us through creating a sample code base, creating a few tests to run against the code base, and configuring Grunt to generate coverage reports while running the tests.

1. We'll start by installing the packages that contain the `blanket`, `mocha-test`, and `contrib-copy` plugins by following the instructions provided in the *Installing a plugin* recipe in Chapter 1, *Getting Started with Grunt*.
2. Then, we can create a JavaScript source file that contains and exports two functions, only one of which we'll be testing. Let's create the file `src/main.js` in our project directory with the following contents:

```
module.exports = {
  square: function (x) {
    return x * x;
  },
  cube: function (x) {
    return x * x * x;
  }
}
```

3. Now, we can write a simple set of tests, using the Mocha framework, to test the `square` method. We will also have to import the `square` method and the `assert` library into our suite of tests, since the `mochaTest` task does not do this automatically. Let's create the file `tests/main.js` in the project directory with the following contents:

```
var assert = require('assert');
var square = require('../src/main').square;

describe('Square method', function() {
  it('returns 4 for 2', function () {
    assert.equal(square(2), 4);
  });
  it('returns 9 for 3', function () {
    assert.equal(square(3), 9);
  });
  it('returns 16 for 4', function () {
    assert.equal(square(4), 16);
  });
});
```

4. With our code base now in place, we can set up its **instrumentation** by adding the following `blanket` task to our configuration:

```
blanket: {
  main: {
    src: 'src/',
    dest: 'coverage/src/'
  }
}
```

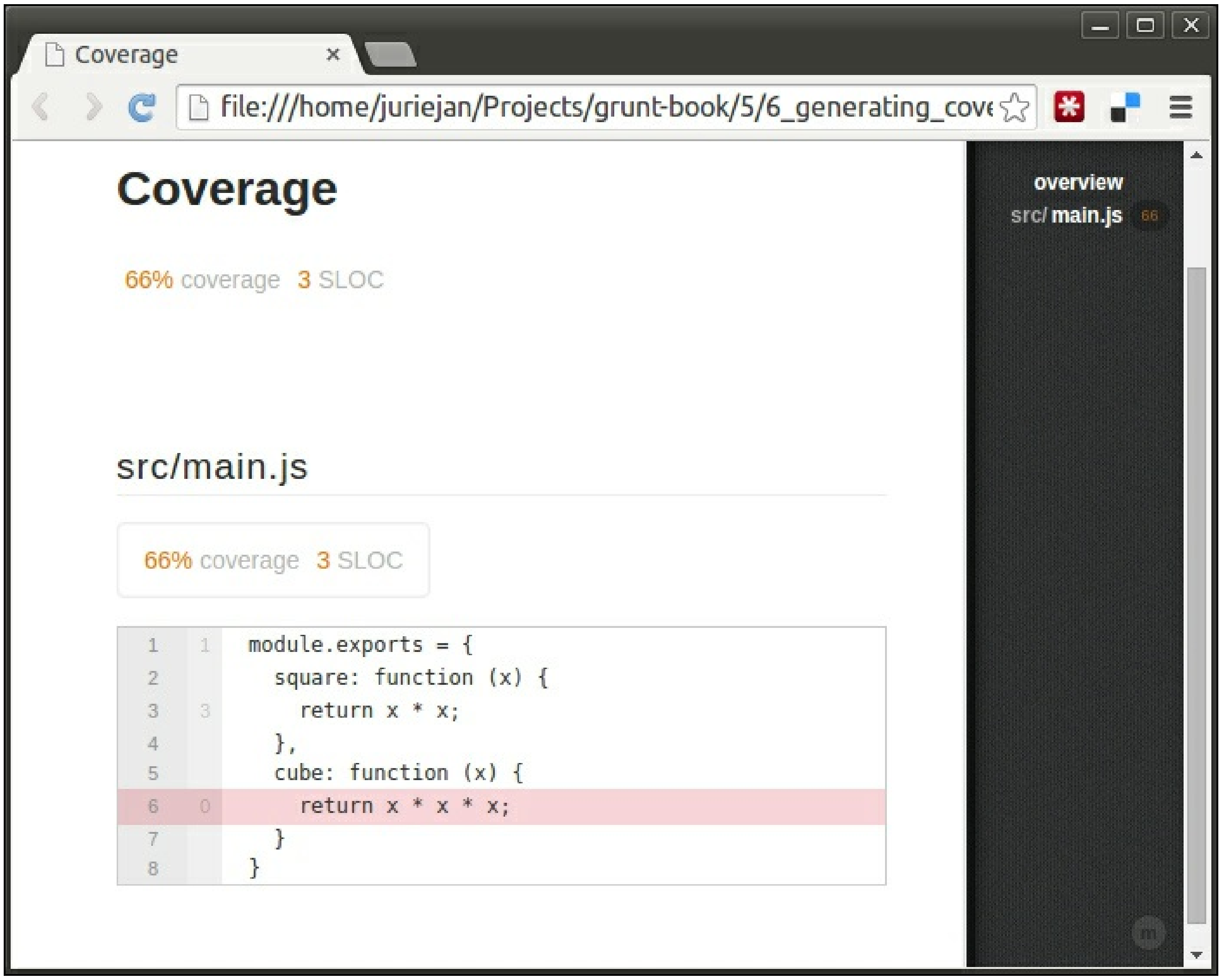
5. In order for our tests to make use of the instrumented version of our code base, we should now also add the following `copy` task to our configuration, which will copy the tests to a position where it can access the instrumented code base, just as it would have accessed the regular code base:

```
copy: {
  tests: {
    src: 'tests/main.js',
    dest: 'coverage/tests/main.js'
  }
}
```

6. To tie it all together, we add the following `mochaTest` task to our configuration, which will run the tests in the position where they have access to the instrumented code base, indicating that it should report the results in the `html-cov` format, and finally save it into the `result.html` file:

```
mochaTest: {
  main: {
    options: {
      quiet: true,
      reporter: 'html-cov',
      captureFile: 'result.html'
    },
    src: 'coverage/tests/main.js'
  }
}
```

7. We can now test our setup by running the `grunt blanket copy mochaTest` command, which should produce a file called `result.html` in our project directory that looks like this:


Coverage
file:///home/juriejan/Projects/grunt-book/5/6_generating_cove
overview
src/ main.js 66

Coverage

66% coverage 3 SLOC

src/main.js

66% coverage 3 SLOC

```
1  module.exports = {
     square: function (x) {
3      return x * x;
     },
     cube: function (x) {
0      return x * x * x;
     }
   }
```

There's more...

The combination of the plugins discussed in this recipe can result in a myriad of configurations, so we'll just focus on the most common requirement of reporting results in the **LCOV** format.

Outputting coverage results in the LCOV format

The LCOV format for code coverage results is quite popular among services that consume them for reporting and analysis purposes.

Tip

The **Coveralls** service is a good example of a service to which you can send your produced LCOV results to keep track of its history and provide a more graphical representation of them.

For more information refer to http://coveralls.io/.

The following steps take us through installing a LCOV reporter for Mocha and altering our configuration to make use of it.

1. We'll start by installing the package that contains the `mocha-lcov-reporter` plugin by following the instructions provided in the *Installing a plugin* recipe in Chapter 1, *Getting Started with Grunt*.
2. Now, we can alter the configuration of our `mochaTest` task to output the results using the newly installed reporter:

   ```
   mochaTest: {
     main: {
       options: {
         quiet: true,
         reporter: 'mocha-lcov-reporter',
         captureFile: 'result.lcov'
       },
       src: 'coverage/tests/main.js'
     }
   }
   ```

3. We can now test our setup by running the `grunt blanket copy mochaTest` command. This should produce a file called `result.lcov` in our project directory, which is ready to be provided to one of the many available services.

Generating coverage reports for client-side code using Mocha and Blanket

In this recipe, we'll make use of the `blanket-mocha (0.4.1)` plugin to run our automated **Mocha** tests in a **PhantomJS** environment, generating **coverage reports** for the source code they run against using the **Blanket.js** library, and comparing the results against a specified threshold.

Getting ready

In this example, we'll work with the basic project structure we created in the *Installing Grunt on a project* recipe in Chapter 1, *Getting Started with Grunt*. Be sure to refer to it if you are not yet familiar with its contents.

How to do it...

The following steps take us through creating a sample code base, creating some tests to run against it, and configuring Grunt to generate coverage reports and compare the results against a threshold.

1. We'll start by installing the package that contains the `blanket-mocha` plugin by following the instructions provided in the *Installing a plugin* recipe in Chapter 1, *Getting Started with Grunt*.
2. Then, we'll create a simple JavaScript source file in our project directory, which contains two functions, one of which we'll be testing. Let's call it `main.js` and add the following content to it:

   ```
   function square(x) {
     return x * x;
   }
   function cube(x) {
     return x * x * x;
   }
   ```

3. Now, we can write a simple suite of tests, using the Mocha and Expect.js frameworks, for the `square` method. Let's create a file called `tests.js` in the project directory with the following contents:

   ```
   describe('Square method', function() {
     it('returns 4 for 2', function () {
       expect(square(2)).to.be(4);
     });
     it('returns 9 for 3', function () {
       expect(square(3)).to.be(9);
     });
     it('returns 16 for 4', function () {
       expect(square(4)).to.be(16);
     });
   });
   ```

4. Due to the tests being run inside a browser and the `blanket_mocha` plugin not automatically including it, we'll have to download the Mocha library and style sheet into the project directory.

   ### Tip

   At the time of writing this, the Mocha library and its accompanying style sheet could be downloaded from the following links:

https://github.com/visionmedia/mocha/raw/master/mocha.js

https://github.com/visionmedia/mocha/raw/master/mocha.css

5. Seeing as the Mocha framework does not include the Expect.js assertion library by default, we also need to download it into our project directory.

   ### Tip

   At the time of writing this, the Expect.js library could be downloaded from the following link:

   https://github.com/LearnBoost/expect.js/raw/master/index.js

   Note that for our current example, we'll download the `index.js` file mentioned, and change its filename to `expect.js`.

6. We'll also have to manually include the Blanket.js client-side library in our testing setup, which means that we'll have to download it into our project directory.

   ### Tip

   At the time of writing this, the Blanket.js client-side library could be downloaded from the following link:

   https://github.com/alex-seville/blanket/raw/master/dist/qunit/blanket.js

7. In order to tie the Mocha and Blanket.js libraries together, we'll also have to include an adapter, which should also be downloaded into our project directory.

   ### Tip

   At the time of writing this, the Mocha-to-Blanket.js adapter could be downloaded from the following link:

   https://github.com/ModelN/grunt-blanket-mocha/raw/master/support/mocha-blanket.js

8. Due to the fact that the `blanket_mocha` task requires the Blanket.js results to be fed back into Grunt, we need to download the appropriate reporter into our project directory.

   ### Tip

At the time of writing this, the appropriate Blanket.js reporter could be downloaded from the following link:

https://github.com/ModelN/grunt-blanket-mocha/raw/master/support/grunt-reporter.js

9. To bring together all the parts we set up, we now need to create a testing environment that loads our code base, tests, and all the required libraries. Let's create the `test.html` file in our project directory with the following contents:

```
<html>
  <head>
    <meta charset="utf-8">
    <title>Mocha Tests</title>
    <link rel="stylesheet" href="mocha.css" />
  </head>
  <body>
    <div id="mocha"></div>
    <script src="expect.js"></script>
    <script src="mocha.js"></script>
    <script src="main.js" data-cover></script>
    <script src="blanket.js"></script>
    <script src="mocha-blanket.js"></script>
    <script>
      mocha.setup('bdd');
      if (window.PHANTOMJS) {
        blanket.options('reporter', 'grunt-reporter.js');
      }
    </script>
    <script src="tests.js"></script>
  </body>
</html>
```

Tip

Note that the `data-cover` attribute provided in the `script` tag imports the `main.js` code base. This attribute should be included in every script tag that imports a file for which coverage reports should be generated.

10. With our code base, tests, and testing environment ready, we can now add the following `blanket_mocha` task to our configuration:

```
blanket_mocha: {
  main: {
    src: ['test.html'],
    options: {
      threshold: 70
```

```
        }
    }
}
```

Tip

We make use of the `threshold` option here to indicate the minimum code coverage percentage that should be achieved for each file. If the final percentage is lower than this threshold, the task will return with a failure.

11. We can then run the task using the `grunt blanket_mocha` command, which will produce output informing us of the test and coverage results, similar to the following:

```
Running "blanket_mocha:main" (blanket_mocha) task
Testing: test.html

  3 passing (3ms)

Per-File Coverage Results: (70% minimum)
PASS : 1 files passed coverage

Unit Test Results: 3 specs passed! (0.00s)
>> No issues found.
```

There's more...

The `blanket-mocha` task provides us with several useful options that can be used in conjunction with its basic test running and coverage reporting features. We'll look at specifying a success threshold for the global average, specifying success thresholds for particular files, and specifying success thresholds for particular modules.

Specifying a success threshold for the global average

If we'd like to set a threshold that the average coverage percentage among all the files in the targeted source code should surpass, we can do so using the `globalThreshold` option as shown in the following example:

```
blanket_mocha: {
  main: {
    src: ['test.html'],
    options: {
      threshold: 70,
      globalThreshold: 70
    }
  }
}
```

Specifying success thresholds for particular files

There's also the option of specifying the thresholds that particular files should adhere to using the `customThreshold` option as shown in the following example:

```
blanket_mocha: {
  main: {
    src: ['test.html'],
    options: {
      threshold: 70,
      customThreshold: {
        'main.js': 70
      }
    }
  }
}
```

Specifying success thresholds for particular modules

In case we'd like to specify a threshold for the average coverage percentage that modules should adhere to, we can make use of the `modulePattern`, moduleThreshold, and customModuleThreshold options.

The modulePattern option takes a **regular expression** of which the first grouping will be used to determine the names of modules. The moduleThreshold option is used to indicate the average threshold that all identified modules should adhere to and the customModuleThreshold option can be used to specify the threshold of each particular module.

The following example identifies modules by the first directory name after the src directory in the filename checks whether the average code coverage percentage of all the modules is above 70 and that the code coverage percentage of the one module is above 60:

```
blanket_mocha: {
  main: {
    src: ['test.html'],
    options: {
      threshold: 70,
      modulePattern: './src/(.*?)/',
      moduleThreshold: 70,
      customModuleThreshold: {
        one: 60
      }
    }
  }
}
```

Generating coverage reports for client-side code using QUnit and Istanbul

In this recipe, we'll make use of the `qunit-istanbul (0.4.5)` plugin to run our automated **QUnit** tests in a **PhantomJS** environment, generating **coverage reports** for the source code they run against using the **Istanbul** library, and comparing the results against a specified threshold.

Getting ready

In this example, we'll work with the basic project structure we created in the *Installing Grunt on a project* recipe in Chapter 1, *Getting Started with Grunt*. Be sure to refer to it if you are not yet familiar with its contents.

How to do it...

The following steps take us through creating a sample code base, creating a few tests to run against it, and configuring Grunt to generate coverage reports and compare the results to a threshold.

1. We'll start by installing the package that contains the `qunit-istanbul` plugin by following the instructions provided in the *Installing a plugin* recipe in Chapter 1, *Getting Started with Grunt*.
2. Then, we'll create a simple JavaScript source file in our project directory that contains two functions, one of which we'll be testing. Let's call it `main.js` and provide it with the following contents:

   ```
   function square(x) {
     return x * x;
   }
   function cube(x) {
     return x * x * x;
   }
   ```

3. Now, we can write a simple set of tests, using the QUnit framework, for the `square` method. Let's create a file called `tests.js` in the project directory with the following contents:

   ```
   QUnit.test("Square method functionality", function(assert) {
     assert.equal(square(2), 4);
     assert.equal(square(3), 9);
     assert.equal(square(4), 16);
   });
   ```

4. Due to the tests being run inside a browser and the `qunit-istanbul` plugin not automatically including it, we'll have to download the QUnit library and style sheet into the project directory.

   ### Tip

   At the time of writing this, the QUnit library and its accompanying style sheet can be found at the following links:

   http://code.jquery.com/qunit/qunit-1.15.0.js

   http://code.jquery.com/qunit/qunit-1.15.0.css

5. To bring together all the parts we set up in the previous steps, we now need to

create a testing environment that loads our code base, tests, and all the required libraries. Let's create the `test.html` file in our project directory with the following contents:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>QUnit basic example</title>
    <link rel="stylesheet" href="qunit-1.15.0.css">
    <script src="qunit-1.15.0.js"></script>
    <script src="main.js"></script>
    <script src="tests.js"></script>
  </head>
  <body>
    <div id="qunit"></div>
    <div id="qunit-fixture"></div>
  </body>
</html>
```

6. With the code base, tests, and testing environment now in place, we can add the following `qunit` task to our configuration:

```
qunit: {
  main: {
    options: {
      coverage: {
        src: 'main.js',
        instrumentedFiles: 'temp/'
      }
    },
    src: 'test.html'
  }
}
```

Tip

Note that all the options provided by the `qunit` task can be used in the `options` object of the `istanbul` task. Only the options contained in the `coverage` part of the `options` object are used to indicate the behavior unique to this plugin.

The `src` option is used to indicate the source files that we'd like to cover in our coverage report. This option can also be set to an array to indicate more files and can also make use of the standard Grunt globbing patterns.

The `instrumentedFiles` option is used to indicate the temporary directory that will contain the instrumented files during the time that the tests are run. Note that this directory is automatically created and destroyed.

7. We can then run the task using the `grunt qunit` command, which should produce output informing us of the test and coverage results, similar to the following:

```
Running "qunit:main" (qunit) task
Testing test.html .OK
>> 3 assertions passed (20ms)
>> Coverage:
>> -  Lines: 75%
>> -  Statements: 75%
>> -  Functions: 50%
>> -  Branches: 100%
```

There's more...

The `qunit-istanbul` task provides us with several useful options that can be used in conjunction with its basic test running and coverage reporting features. We'll look at how to specify a report output destination and coverage thresholds at various levels.

Specifying a report output destination

If we'd like to save the coverage report results to files, we can do so by providing a directory name to any of the highlighted options in the following example, each of which saves the results in the particular format mentioned in its name:

```
qunit: {
  main: {
    options: {
      coverage: {
        src: 'main.js',
        instrumentedFiles: 'temp/',
        htmlReport: 'html/',
        coberturaReport: 'corb/',
        lcovReport: 'lcov/',
        cloverReport: 'clover/'
      }
    },
    src: 'test.html'
  }
}
```

Specifying coverage thresholds at various levels

In case we'd like to specify the code coverage percentage thresholds at either the line, statement, function, or branch level, we can do so by using any of the appropriately named options highlighted in the following example:

```
qunit: {
  main: {
    options: {
      coverage: {
        src: 'main.js',
        instrumentedFiles: 'temp/',
        linesThresholdPct: 50,
        statementsThresholdPct: 60,
        functionsThresholdPct: 70,
        branchesThresholdPct: 80
      }
    },
    src: 'test.html'
  }
}
```

Chapter 6. Deployment Preparations

In this chapter, we will cover the following recipes:

- Minifying HTML
- Minifying CSS
- Optimizing images
- Linting JavaScript code
- Uglifying JavaScript code
- Setting up RequireJS

Introduction

Once our web application is built and its stability ensured, we can start preparing it for deployment to its intended market. This will mainly involve the optimization of the assets that make up the application. Optimization in this context mostly refers to compression of one kind or another, some of which might lead to performance increases too.

The focus on compression is primarily due to the fact that the smaller the asset, the faster it can be transferred from where it is hosted to a user's web browser. This leads to a much better user experience, and can sometimes be essential to the functioning of an application.

Minifying HTML

In this recipe, we make use of the `contrib-htmlmin (0.3.0)` plugin to decrease the size of some **HTML** documents by **minifying** them.

Getting ready

In this example, we'll work with the basic project structure we created in the *Installing Grunt on a project* recipe in Chapter 1, *Getting Started with Grunt*. Be sure to refer to it if you are not yet familiar with its contents.

How to do it...

The following steps take us through creating a sample HTML document and configuring a task that minifies it:

1. We'll start by installing the package that contains the `contrib-htmlmin` plugin as per the instructions provided in the *Installing a plugin* recipe in Chapter 1, *Getting Started with Grunt*.
2. Next, we'll create a simple HTML document called `index.html` in the `src` directory, which we'd like to minify, and add the following content in it:

```
<html>
  <head>
    <title>Test Page</title>
  </head>
  <body>
    <!-- This is a comment! -->
    <h1>This is a test page.</h1>
  </body>
</html>
```

3. Now, we'll add the following `htmlmin` task to our configuration, which indicates that we'd like to have the white space and comments removed from the `src/index.html` file, and that we'd like the result to be saved in the `dist/index.html` file:

```
htmlmin: {
  dist: {
    src: 'src/index.html',
    dest: 'dist/index.html',
    options: {
      removeComments: true,
      collapseWhitespace: true
    }
  }
}
```

Tip

The `removeComments` and `collapseWhitespace` options are used as examples here, as using the default `htmlmin` task will have no effect. Other minification options can be found at the following URL:

https://github.com/kangax/html-minifier#options-quick-reference

4. We can now run the task using the `grunt htmlmin` command, which should produce output similar to the following:

```
Running "htmlmin:dist" (htmlmin) task
Minified dist/index.html 147 B → 92 B
```

5. If we now take a look at the `dist/index.html` file, we will see that all white space and comments have been removed:

```
<html>
  <head>
    <title>Test Page</title>
  </head>
  <body>
    <h1>This is a test page.</h1>
  </body>
</html>
```

Minifying CSS

In this recipe, we'll make use of the `contrib-cssmin (0.10.0)` plugin to decrease the size of some **CSS** documents by **minifying** them.

Getting ready

In this example, we'll work with the basic project structure we created in the *Installing Grunt on a project* recipe in Chapter 1, *Getting Started with Grunt*. Be sure to refer to it if you are not yet familiar with its contents.

How to do it...

The following steps take us through creating a sample CSS document and configuring a task that minifies it.

1. We'll start by installing the package that contains the `contrib-cssmin` plugin as per the instructions provided in the *Installing a plugin* recipe in Chapter 1, *Getting Started with Grunt*.
2. Then, we'll create a simple CSS document called `style.css` in the `src` directory, which we'd like to minify, and provide it with the following contents:

```
body {
  /* Average body style */
  background-color: #ffffff;
  color: #000000; /*! Black (Special) */
}
```

3. Now, we'll add the following `cssmin` task to our configuration, which indicates that we'd like to have the `src/style.css` file compressed, and have the result saved to the `dist/style.min.css` file:

```
cssmin: {
  dist: {
    src: 'src/style.css',
    dest: 'dist/style.min.css'
  }
}
```

4. We can now run the task using the `grunt cssmin` command, which should produce the following output:

```
Running "cssmin:dist" (cssmin) task
File dist/style.css created: 55 B → 38 B
```

5. If we take a look at the `dist/style.min.css` file that was produced, we will see that it has the compressed contents of the original `src/style.css` file:

```
body{background-color:#fff;color:#000;/*! Black (Special) */}
```

There's more...

The `cssmin` task provides us with several useful options that can be used in conjunction with its basic compression feature. We'll look at prefixing a banner, removing special comments, and reporting gzipped results.

Prefixing a banner

In the case that we'd like to automatically include some information about the compressed result in the resulting CSS file, we can do so in a banner. A banner can be prepended to the result by supplying the desired banner content to the `banner` option, as shown in the following example:

```
cssmin: {
  dist: {
    src: 'src/style.css',
    dest: 'dist/style.min.css',
    options: {
      banner: '/* Minified version of style.css */'
    }
  }
}
```

Removing special comments

Comments that should not be removed by the minification process are called special comments and can be indicated using the "`/*! comment */`" markers. By default, the `cssmin` task will leave all special comments untouched, but we can alter this behavior by making use of the `keepSpecialComments` option.

The `keepSpecialComments` option can be set to either the `*`, `1`, or `0` value. The `*` value is the default and indicates that all special comments should be kept, `1` indicates that only the first comment that is found should be kept, and `0` indicates that none of them should be kept. The following configuration will ensure that all comments are removed from our minified result:

```
cssmin: {
  dist: {
    src: 'src/style.css',
    dest: 'dist/style.min.css',
    options: {
      keepSpecialComments: 0
    }
  }
}
```

Reporting on gzipped results

Reporting is useful to see exactly how well the `cssmin` task has compressed our CSS files. By default, the size of the targeted file and minified result will be displayed, but if we'd also like to see the gzipped size of the result, we can set the `report` option to `gzip`, as shown in the following example:

```
cssmin: {
  dist: {
    src: 'src/main.css',
    dest: 'dist/main.css',
    options: {
      report: 'gzip'
    }
  }
}
```

Optimizing images

In this recipe, we'll make use of the `contrib-imagemin (0.9.4)` plugin to decrease the size of images by compressing them as much as possible without compromising on their quality. This plugin also provides a plugin framework of its own, which is discussed at the end of this recipe.

Getting ready

In this example, we'll work with the basic project structure we created in the *Installing Grunt on a project* recipe in Chapter 1, *Getting Started with Grunt*. Be sure to refer to it if you are not yet familiar with its contents.

How to do it...

The following steps take us through configuring a task that will compress an image for our project.

1. We'll start by installing the package that contains the `contrib-imagemin` plugin as per the instructions provided in the *Installing a plugin* recipe in Chapter 1, *Getting Started with Grunt*.
2. Next, we can ensure that we have an image called `image.jpg` in the `src` directory on which we'd like to perform optimizations.

   ### Tip

   For the rest of this example, we'll make use of the sample image provided with the sample code that accompanies this recipe.

3. Now, we'll add the following `imagemin` task to our configuration and indicate that we'd like to have the `src/image.jpg` file optimized, and have the result saved to the `dist/image.jpg` file:

   ```
   imagemin: {
     dist: {
       src: 'src/image.jpg',
       dest: 'dist/image.jpg'
     }
   }
   ```

4. We can then run the task using the `grunt imagemin` command, which should produce the following output:

   ```
   Running "imagemin:dist" (imagemin) task
   Minified 1 image (saved 13.36 kB)
   ```

5. If we now take a look at the `dist/image.jpg` file, we will see that its size has decreased without any impact on the quality.

There's more...

The `imagemin` task provides us with several options that allow us to tweak its optimization features. We'll look at how to adjust the **PNG** compression level, disable the progressive **JPEG** generation, disable the interlaced **GIF** generation, specify **SVGO** plugins to be used, and use the `imagemin` plugin framework.

Adjusting the PNG compression level

The compression of a PNG image can be increased by running the compression algorithm on it multiple times. By default, the compression algorithm is run 16 times. This number can be changed by providing a number from `0` to `7` to the `optimizationLevel` option. The `0` value means that the compression is effectively disabled and `7` indicates that the algorithm should run 240 times. In the following configuration we set the compression level to its maximum:

```
imagemin: {
  dist: {
    src: 'src/image.png',
    dest: 'dist/image.png',
    options: {
      optimizationLevel: 7
    }
  }
}
```

Disabling the progressive JPEG generation

Progressive JPEGs are compressed in multiple passes, which allows a low-quality version of them to quickly become visible and increase in quality as the rest of the image is received. This is especially helpful when displaying images over a slower connection.

By default, the `imagemin` plugin will generate JPEG images in the progressive format, but this behavior can be disabled by setting the `progressive` option to `false`, as shown in the following example:

```
imagemin: {
  dist: {
    src: 'src/image.jpg',
    dest: 'dist/image.jpg',
    options: {
      progressive: false
    }
  }
}
```

Disabling the interlaced GIF generation

An interlaced GIF is the equivalent of a progressive JPEG in that it allows the contained image to be displayed at a lower resolution before it has been fully downloaded, and increases in quality as the rest of the image is received.

By default, the `imagemin` plugin will generate GIF images in the interlaced format, but this behavior can be disabled by setting the `interlaced` option to `false`, as shown in the following example:

```
imagemin: {
  dist: {
    src: 'src/image.gif',
    dest: 'dist/image.gif',
    options: {
      interlaced: false
    }
  }
}
```

Specifying SVGO plugins to be used

When optimizing SVG images, the SVGO library is used by default. This allows us to specify the use of various plugins provided by the SVGO library that each performs a specific function on the targeted files.

Tip

Refer to the following URL for more detailed instructions on how to use the svgo plugins options and the SVGO library:

https://github.com/sindresorhus/grunt-svgmin#available-optionsplugins

Most of the plugins in the library are enabled by default, but if we'd like to specifically indicate which of these should be used, we can do so using the `svgoPlugins` option. Here, we can provide an array of objects, where each contain a property with the name of the plugin to be affected, followed by a `true` or `false` value to indicate whether it should be activated. The following configuration disables three of the default plugins:

```
 imagemin: {
    dist: {
      src: 'src/image.svg',
      dest: 'dist/image.svg',
      options: {
        svgoPlugins: [
          {removeViewBox:false},
          {removeUselessStrokeAndFill:false},
          {removeEmptyAttrs:false}
        ]
      }
    }
  }
```

Using the 'imagemin' plugin framework

In order to provide support for the various image optimization projects, the `imagemin` plugin has a plugin framework of its own that allows developers to easily create an extension that makes use of the tool they require.

Tip

You can get a list of the available plugin modules for the `imagemin` plugin's framework at the following URL:

https://www.npmjs.com/browse/keyword/imageminplugin

The following steps will take us through installing and making use of the `mozjpeg` plugin to compress an image in our project. These steps start where the main recipe takes off.

1. We'll start by installing the `imagemin-mozjpeg` package using the `npm install imagemin-mozjpeg` command, which should produce the following output:

   ```
   imagemin-mozjpeg@4.0.0 node_modules/imagemin-mozjpeg
   ```

2. With the package installed, we need to import it into our configuration file, so that we can make use of it in our task configuration. We do this by adding the following line at the top of our `Gruntfile.js` file:

   ```
   var mozjpeg = require('imagemin-mozjpeg');
   ```

3. With the plugin installed and imported, we can now change the configuration of our `imagemin` task by adding the `use` option and providing it with the initialized plugin:

   ```
   imagemin: {
   ```

```
  dist: {
    src: 'src/image.jpg',
    dest: 'dist/image.jpg',
    options: {
      use: [mozjpeg()]
    }
  }
}
```

4. Finally, we can test our setup by running the task using the `grunt imagemin` command. This should produce an output similar to the following:

```
Running "imagemin:dist" (imagemin) task
Minified 1 image (saved 9.88 kB)
```

Linting JavaScript code

In this recipe, we'll make use of the `contrib-jshint (0.11.1)` plugin to detect errors and potential problems in our JavaScript code. It is also commonly used to enforce code conventions within a team or project. As can be derived from its name, it's basically a Grunt adaptation for the JSHint tool.

Getting ready

In this example, we'll work with the basic project structure we created in the *Installing Grunt on a project* recipe in Chapter 1, *Getting Started with Grunt*. Be sure to refer to it if you are not yet familiar with its contents.

How to do it...

The following steps take us through creating a sample JavaScript file and configuring a task that will scan and analyze it using the JSHint tool.

1. We'll start by installing the package that contains the `contrib-jshint` plugin as per the instructions provided in the *Installing a plugin* recipe in Chapter 1, *Getting Started with Grunt*.
2. Next, we'll create a sample JavaScript file called `main.js` in the `src` directory, and add the following content in it:

```
sample = 'abc';
console.log(sample);
```

3. With our sample file ready, we can now add the following `jshint` task to our configuration. We'll configure this task to target the sample file and also add a basic option that we require for this example:

```
jshint: {
  main: {
    options: {
      undef: true
    },
    src: ['src/main.js']
  }
}
```

Tip

The `undef` option is a standard JSHint option used specifically for this example and is not required for this plugin to function. Specifying this option indicates that we'd like to have errors raised for variables that are used without being explicitly defined.

4. We can now run the task using the `grunt jshint` command, which should produce output informing us of the problems found in our sample file:

```
Running "jshint:main" (jshint) task

   src/main.js
      1 |sample = 'abc';
         ^ 'sample' is not defined.
      2 |console.log(sample);
         ^ 'console' is not defined.
      2 |console.log(sample);
```

```
                                         ^ 'sample' is not defined.

>> 3 errors in 1 file
```

There's more...

The `jshint` task provides us with several options that allow us to change its general behavior, in addition to how it analyzes the targeted code. We'll look at how to specify standard JSHint options, specify globally defined variables, send reported output to a file, and prevent task failure on JSHint errors.

Specifying standard JSHint options

The `contrib-jshint` plugin provides a simple way to pass all the standard JSHint options from the task's options object to the underlying JSHint tool.

Tip

A list of all the options provided by the JSHint tool can be found at the following URL:

http://jshint.com/docs/options/

The following example adds the `curly` option to the task we created in our main recipe to enforce the use of curly braces wherever they are appropriate:

```
jshint: {
  main: {
    options: {
      undef: true,
      curly: true
    },
    src: ['src/main.js']
  }
}
```

Specifying globally defined variables

Making use of globally defined variables is quite common when working with JavaScript, which is where the `globals` option comes in handy. Using this option, we can define a set of global values that we'll use in the targeted code, so that errors aren't raised when JSHint encounters them.

In the following example, we indicate that the `console` variable should be treated as a global, and not raise errors when encountered:

```
jshint: {
  main: {
    options: {
      undef: true,
      globals: {
        console: true
      }
    },
    src: ['src/main.js']
  }
}
```

Sending reported output to a file

If we'd like to store the resulting output from our JSHint analysis, we can do so by specifying a path to a file that should receive it using the `reporterOutput` option, as shown in the following example:

```
jshint: {
  main: {
    options: {
      undef: true,
      reporterOutput: 'report.dat'
    },
    src: ['src/main.js']
  }
}
```

Preventing task failure on JSHint errors

The default behavior for the `jshint` task is to exit the running Grunt process once a JSHint error is encountered in any of the targeted files. This behavior becomes especially undesirable if you'd like to keep watching files for changes, even when an error has been raised.

In the following example, we indicate that we'd like to keep the process running when errors are encountered by giving the `force` option a `true` value:

```
jshint: {
  main: {
    options: {
      undef: true,
      force: true
    },
    src: ['src/main.js']
  }
}
```

Uglifying JavaScript Code

In this recipe, we'll make use of the `contrib-uglify (0.8.0)` plugin to compress and mangle some files containing JavaScript code.

For the most part, the process of uglifying just removes all the unnecessary characters and shortens variable names in a source code file. This has the potential to dramatically reduce the size of the file, slightly increase performance, and make the inner workings of your publicly available code a little more obscure.

Getting ready

In this example, we'll work with the basic project structure we created in the *Installing Grunt on a project* recipe in Chapter 1, *Getting Started with Grunt*. Be sure to refer to it if you are not yet familiar with its contents.

How to do it...

The following steps take us through creating a sample JavaScript file and configuring a task that will uglify it.

1. We'll start by installing the package that contains the `contrib-uglify` plugin as per the instructions provided in the *Installing a plugin* recipe in Chapter 1, *Getting Started with Grunt*.
2. Then, we can create a sample JavaScript file called `main.js` in the `src` directory, which we'd like to uglify, and provide it with the following contents:

   ```
   var main = function () {
     var one = 'Hello' + ' ';
     var two = 'World';

     var result = one + two;

     console.log(result);
   };
   ```

3. With our sample file ready, we can now add the following `uglify` task to our configuration, indicating the sample file as the target and providing a destination output file:

   ```
   uglify: {
     main: {
       src: 'src/main.js',
       dest: 'dist/main.js'
     }
   }
   ```

4. We can now run the task using the `grunt uglify` command, which should produce output similar to the following:

   ```
   Running "uglify:main" (uglify) task
   >> 1 file created.
   ```

5. If we now take a look at the resulting `dist/main.js` file, we should see that it contains the uglified contents of the original `src/main.js` file.

There's more...

The `uglify` task provides us with several options that allow us to change its general behavior and see how it uglifies the targeted code. We'll look at specifying standard UglifyJS options, generating source maps, and wrapping generated code in an enclosure.

Specifying standard UglifyJS options

The underlying UglifyJS tool can provide a set of options for each of its separate functional parts. These parts are the mangler, compressor, and beautifier. The `contrib-plugin` allows passing options to each of these parts using the `mangle`, `compress`, and `beautify` options.

Tip

The available options for each of the mangler, compressor, and beautifier parts can be found at each of following URLs (listed in the order mentioned):

https://github.com/mishoo/UglifyJS2#mangler-options

https://github.com/mishoo/UglifyJS2#compressor-options

https://github.com/mishoo/UglifyJS2#beautifier-options

The following example alters the configuration of the main recipe to provide a single option to each of these parts:

```
uglify: {
  main: {
    src: 'src/main.js',
    dest: 'dist/main.js',
    options: {
      mangle: {
        toplevel: true
      },
      compress: {
        evaluate: false
      },
      beautify: {
        semicolons: false
      }
    }
  }
}
```

Generating source maps

As code gets mangled and compressed, it becomes effectively unreadable to humans, and therefore, nearly impossible to debug. For this reason, we are provided with the option of generating a source map when uglifying our code.

The following example makes use of the `sourceMap` option to indicate that we'd like to have a source map generated along with our uglified code:

```
uglify: {
  main: {
    src: 'src/main.js',
    dest: 'dist/main.js',
    options: {
      sourceMap: true
    }
  }
}
```

Running the altered task will now, in addition to the `dist/main.js` file with our uglified source, generate a source map file called `main.js.map` in the same directory as the uglified file.

Wrapping generated code in an enclosure

When building your own JavaScript code modules, it's usually a good idea to have them wrapped in a wrapper function to ensure that you don't pollute the global scope with variables that you won't be using outside of the module itself.

For this purpose, we can use the `wrap` option to indicate that we'd like to have the resulting uglified code wrapped in a wrapper function, as shown in the following example:

```
uglify: {
  main: {
    src: 'src/main.js',
    dest: 'dist/main.js',
    options: {
      wrap: true
    }
  }
}
```

If we now take a look at the result `dist/main.js` file, we should see that all the uglified contents of the original file are now contained within a wrapper function.

Setting up RequireJS

In this recipe, we'll make use of the `contrib-requirejs (0.4.4)` plugin to package the modularized source code of our web application into a single file.

For the most part, this plugin just provides a wrapper for the **RequireJS** tool. RequireJS provides a framework to modularize JavaScript source code and consume those modules in an orderly fashion. It also allows packaging an entire application into one file and importing only the modules that are required while keeping the module structure intact.

Tip

To see the packaged application created in this recipe in action, please refer to the sample code provided for this recipe. It includes a basic development server setup as per the *Setting up a basic web server* recipe in Chapter 1, *Getting Started with Grunt* along with the required libraries and a sample `index.html` file that consumes the generated application bundle file.

Getting ready

In this example, we'll work with the basic project structure we created in the *Installing Grunt on a project* recipe in Chapter 1, *Getting Started with Grunt*. Be sure to refer to it if you are not yet familiar with its contents.

How to do it...

The following steps take us through creating some files for a sample application and setting up a task that bundles them into one file.

1. We'll start by installing the package that contains the `contrib-requirejs` plugin as per the instructions provided in the *Installing a plugin* recipe in Chapter 1, *Getting Started with Grunt*.
2. First, we'll need a file that will contain our RequireJS configuration. Let's create a file called `config.js` in the `src` directory and add the following content in it:

   ```
   require.config({
     baseUrl: 'app'
   });
   ```
3. Secondly, we'll create a sample module that we'd like to use in our application. Let's create a file called `sample.js` in the `src/app` directory and add the following content in it:

   ```
   define(function (require) {
     return function () {
       console.log('Sample Module');
     }
   });
   ```
4. Lastly, we'll need a file that will contain the main entry point for our application, and also makes use of our sample module. Let's create a file called `main.js` in the `src/app` directory and add the following content in it:

   ```
   require(['sample'], function (sample) {
     sample();
   });
   ```
5. Now that we've got all the necessary files required for our sample application, we can setup a `requirejs` task that will bundle it all into one file:

   ```
   requirejs: {
     app: {
       options: {
         mainConfigFile: 'src/config.js',
         name: 'main',
         out: 'www/js/app.js'
       }
     }
   }
   ```

Tip

The `mainConfigFile` option points out the configuration file that will determine the behavior of RequireJS.

The `name` option indicates the name of the module that contains the application entry point. In the case of this example, our application entry point is contained in the `app/main.js` file, and `app` is the base directory of our application in the `src/config.js` file. This translates the `app/main.js` filename into the `main` module name.

The `out` option is used to indicate the file that should receive the result of the bundled application.

6. We can now run the task using the `grunt requirejs` command, which should produce output similar to the following:

   ```
   Running "requirejs:app" (requirejs) task
   ```

7. We should now have a file named `app.js` in the `www/js` directory that contains our entire sample application.

There's more...

The `requirejs` task provides us with all the underlying options provided by the RequireJS tool. We'll look at how to use these exposed options and generate a source map.

Using RequireJS optimizer options

The RequireJS optimizer is quite an intricate tool, and therefore, provides a large number of options to tweak its behavior. The `contrib-requirejs` plugin allows us to easily set any of these options by just specifying them as options of the plugin itself.

Tip

A list of all the available configuration options for the RequireJS build system can be found in the example configuration file at the following URL:

https://github.com/jrburke/r.js/blob/master/build/example.build.js

The following example indicates that the **UglifyJS2** optimizer should be used instead of the default **UglifyJS** optimizer by using the `optimize` option:

```
requirejs: {
  app: {
    options: {
      mainConfigFile: 'src/config.js',
      name: 'main',
      out: 'www/js/app.js',
      optimize: 'uglify2'
    }
  }
}
```

Generating a source map

When the source code is bundled into one file, it becomes somewhat harder to debug, as you now have to trawl through miles of code to get to the point you're actually interested in.

A source map can help us with this issue by relating the resulting bundled file to the modularized structure it is derived from. Simply put, with a source map, our debugger will display the separate files we had before, even though we're actually using the bundled file.

The following example makes use of the `generateSourceMap` option to indicate that we'd like to generate a source map along with the resulting file:

```
requirejs: {
  app: {
    options: {
      mainConfigFile: 'src/config.js',
      name: 'main',
      out: 'www/js/app.js',
      optimize: 'uglify2',
      preserveLicenseComments: false,
      generateSourceMaps: true
    }
  }
}
```

Tip

In order to use the `generateSourceMap` option, we have to indicate that UglifyJS2 is to be used for optimization, by setting the `optimize` option to `uglify2`, and that license comments should not be preserved, by setting the `preserveLicenseComments` option to `false`.

Chapter 7. Deploying to the End User

In this chapter, we will cover the following recipes:

- Deploying to Rackspace Cloud Files
- Deploying to AWS S3
- Deploying over FTP
- Deploying over SFTP
- Deploying to GitHub Pages
- Invalidating an AWS CloudFront distribution
- Running commands over SSH

Introduction

Once our web application is built and its assets are optimized for optimal delivery and consumption, it's time to make it available to our intended audience. This primarily involves transferring the assets that make up the application to some form of file hosting system that is dedicated to the delivery of static content over the Internet.

The primary points of focus in deploying the assets of a web application to the Internet are availability, speed, and service integration. Assets should always be available from anywhere in the world, be delivered as fast as possible, and the host system should allow us to easily upload and manage our content.

Deploying to Rackspace Cloud Files

In this recipe, we'll make use of the `cloudfiles (0.3.0)` plugin to upload files to a **Rackspace Cloud Files** container.

The Cloud Files service has the added benefit of providing a **Content Delivery Network** (**CDN**) service to all content hosted on it, by default.

Getting ready

In this example, we'll work with the basic project structure that we created in the *Installing Grunt on a project* recipe in Chapter 1, *Getting Started with Grunt*. Be sure to refer to it if you are not yet familiar with its contents.

The following example will also require a **Rackspace Cloud** account with an API key defined. We'll also need to create a Cloud Files container named `myapp` and configure it to host a **static website**.

Tip

Refer to the following URL for more information on configuring a Cloud Files container to host a static website:

http://docs.rackspace.com/files/api/v1/cf-devguide/content/Create_Static_Website-dle4000.html

How to do it...

The following steps take us through creating a simple HTML document and a task that uploads it to a Cloud Files container:

1. We'll start by installing the package that contains the `cloudfiles` plugin, as per the instructions provided in the *Installing a plugin* recipe in Chapter 1, *Getting Started with Grunt*.
2. Next, we'll create a simple HTML document called `index.html` in the project directory, and provide it with the following contents:

   ```
   <html>
     <head>
       <title>MyApp</title>
     </head>
     <body>
       <h1>This is MyApp.</h1>
     </body>
   </html>
   ```

3. Now, we'll add the following `cloudfiles` task to our configuration, which will indicate that we'd like to have the `index.html` file uploaded to the `myapp` Cloud Files container:

   ```
   cloudfiles: {
     myapp: {
       user: '[username]',
       key: '[api key]',
       region: 'LON',
       upload: [{
         container: 'myapp',
         src: 'index.html',
       }]
     }
   }
   ```

   ### Tip

   The `user` and `key` options are filled with placeholder values to indicate that you should make use of your own Rackspace Cloud account's username and API key.

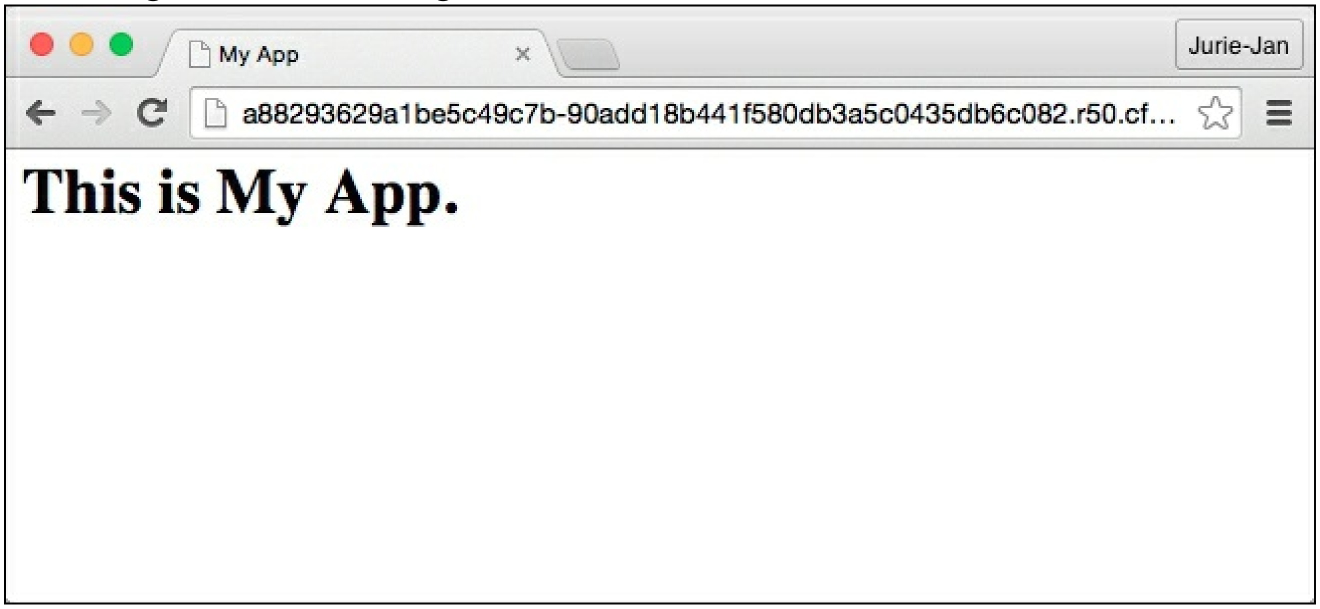
   You'll probably want to have the `user` and `key` options stored in a local file, as opposed to a shared repository. Refer to the *Importing external data* recipe of Chapter 1, *Getting Started with Grunt* for an example of how to import configurations from external files. Also, be sure to exclude files that contain access credentials from your project's code repository.

The `region` option is used to indicate the geographical region of the hosted Cloud Files container. The desired region is indicated on creation of the container. For our example, we created a container in the `LON` region.

4. We can now run the task by using the `grunt cloudfiles` command, which should produce output similar to the following:

```
Running "cloudfiles:myapp" (cloudfiles) task
Uploading into myapp
Syncing files to container: myapp
Uploading index.html to myapp (NEW)
```

5. We can now ensure that the file has been uploaded to our container and is accessible via the Internet. To do this, we'll need to determine the target domain of the container from its settings and navigate to it in our browser. This should look something like the following:


My App
Jurie-Jan
a88293629a1be5c49c7b-90add18b441f580db3a5c0435db6c082.r50.cf...

This is My App.

There's more...

The `cloudfiles` task provides us with several useful options that can be used in conjunction with its basic uploading feature. We'll be looking at uploading the contents of a directory to a destination directory.

Uploading the contents of a directory

In case we'd like to upload the contents of a directory, we can make use of the standard globbing patterns supported by Grunt in the `src` option. We would, however, probably also want to make use of the `stripcomponents` option to remove the leading paths from the directory names of the destination files.

The following example will upload the contents of the `www` directory to the `myapp` container, and strip the first path name from the target files when determining the destination file names:

```
cloudfiles: {
  myapp: {
    user: 'juriejan',
    key: 'c980b9327b823b96dd83b51cdc5cf7dd',
    region: 'LON',
    upload: [{
      container: 'myapp',
      src: 'www/**/*',
      stripcomponents: 1
    }]
  }
}
```

Uploading to a destination directory

In case we'd like to upload files to a specific destination directory on our target container, we can indicate the target directory by using the `dest` option, as shown in the following example:

```
cloudfiles: {
  myapp: {
    user: 'juriejan',
    key: 'c980b9327b823b96dd83b51cdc5cf7dd',
    region: 'LON',
    upload: [{
      container: 'myapp',
      src: 'index.html',
        dest: '/htmlfiles/'
      }]
   }
}
```

Deploying to AWS S3

In this recipe, we'll make use of the `aws-s3 (0.12.3)` plugin to upload files to an **AWS S3 bucket**.

This service doesn't provide a CDN setup by default, but can be easily integrated with **AWS CloudFront** in order to speed up the distribution of the hosted files.

Getting ready

In this example, we'll work with the basic project structure that we created in the *Installing Grunt on a project* recipe in Chapter 1, *Getting Started with Grunt*. Be sure to refer to it if you are not yet familiar with its contents.

In addition to the standard project setup, the following recipe will also require the setup of an **AWS user** with an **AWS access key**.

Tip

Refer to the following URL for details on how to obtain your AWS security credentials:

http://docs.aws.amazon.com/general/latest/gr/getting-aws-sec-creds.html

A bucket with a name in the `[name].myapp` format will also need to be created, with `[name]` being any unique name that you wish to use. The aforementioned user should also have full access granted to the created bucket using the **AWS IAM** interface.

Tip

Refer to the following URL for more information on how to grant a user account access to an S3 bucket:

http://blogs.aws.amazon.com/security/post/Tx3VRSWZ6B3SHAV/Writing-IAM-Policies-How-to-grant-access-to-an-Amazon-S3-bucket

In order for the bucket to behave as required, it should also be configured for **static website hosting**, and `index.html` should be defined as its **index document**.

Tip

Refer to the following URL for more information on how to configure an AWS S3 bucket for static website hosting:

http://docs.aws.amazon.com/AmazonS3/latest/dev/HowDoIWebsiteConfiguration.html

How to do it...

The following steps take us through creating a sample HTML document and configuring a task that uploads it to an AWS S3 bucket:

1. We'll start by installing the package that contains the `aws-s3` plugin, as per the instructions provided in the *Installing a plugin* recipe in Chapter 1, *Getting Started with Grunt*.
2. Next, we'll create a simple HTML document called `index.html` in the project directory, and provide it with the following contents:

```
<html>
  <head>
    <title>MyApp</title>
  </head>
  <body>
    <h1>This is MyApp.</h1>
  </body>
</html>
```

3. Now, we'll add the following `aws_s3` task to our configuration, which indicates that we'd like to upload the `index.html` file to the `gruntbook.myapp` bucket:

```
aws_s3: {
  myapp: {
    options: {
      accessKeyId: '[access key id]',
      secretAccessKey: '[secret access key]',
      region: 'eu-west-1',
      bucket: 'gruntbook.myapp',
    },
    files: [{
      src: 'index.html',
      dest: '/'
    }]
  }
}
```

Tip

The `accessKeyId` and `secretAccessKey` options are filled with placeholder values to indicate that you should make use of your own AWS access credentials.

The `accessKeyId` option should be set to an **access key ID**, generated for your user using the **AWS Identity and Access Management** (**IAM**) console.

The `secretAccessKey` option should be set to the **secret access key** that was generated for the specific access key ID that you specified in the `accessKeyId` option. Note that the secret access key is only displayed on creation of the access key ID, so you won't be able to find it in the IAM console if you didn't save it the first time.

You'll probably want to have the `accessKeyId` and `secretAccessKey` options stored in a local file, as opposed to a shared repository. Refer to the *Importing external data* recipe of Chapter 1, *Getting Started with Grunt* for an example of how to import configurations from external files. Also, be sure to exclude files that contain access credentials from your project's code repository.

The `region` option is used to indicate the geographical region of the hosted bucket. The desired region is indicated on creation of the bucket. For our example, we created a container in the `eu-west-1` region.

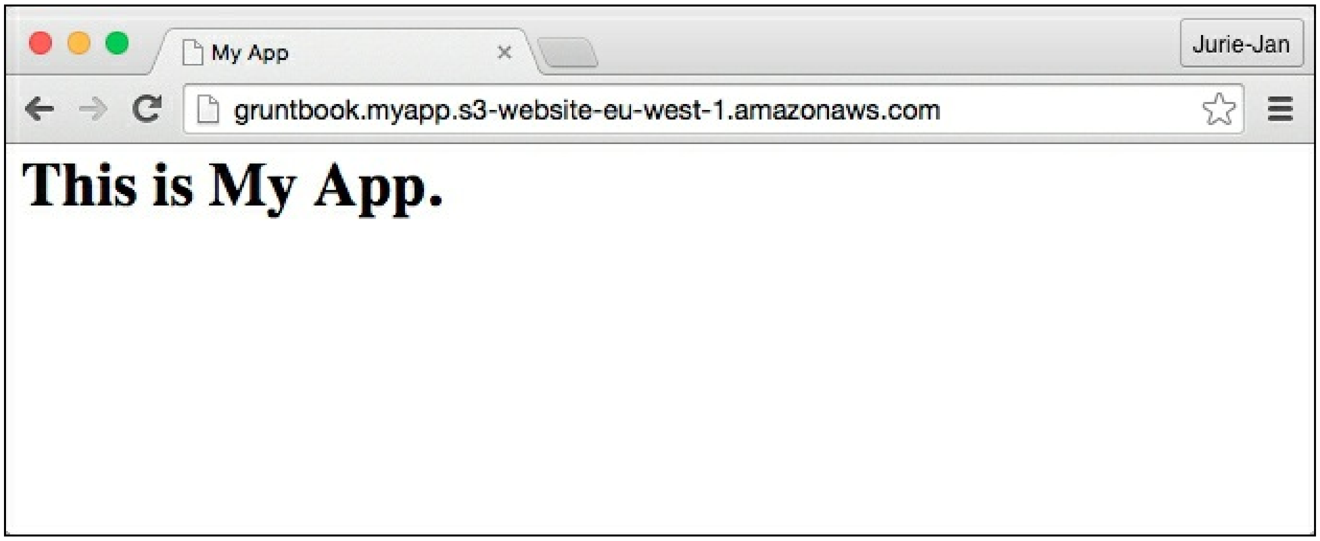
Note that the files configuration for this task supports all the standard Grunt options. You can read more about them at the following URL:

http://gruntjs.com/configuring-tasks#files

4. We can now run the task by using the `grunt aws_s3` command, which should produce output similar to the following:

```
Running "aws_s3:myapp" (aws_s3) task
Uploading to https://s3-eu-west-1.amazonaws.com/gruntbook.myapp/
.
List: (1 objects):
- index.html -> https://s3-eu-west-
1.amazonaws.com/gruntbook.myapp/index.html
```

5. We can now ensure that the file has been uploaded to our container and is accessible via the Internet. To do this, we'll need to determine the endpoint of the bucket from its properties section and navigate to it in our browser. This should look something like the following:



This is My App.

There's more...

The `aws_s3` task provides us with several useful options that can be used in conjunction with its uploading feature. We'll look at how to specify the accessibility of uploaded files and enable concurrent operations.

Specifying the accessibility of uploaded files

Every file uploaded to an AWS S3 bucket has a set of access permissions that indicate who has access to it. If we'd like to indicate a specific set of permissions, we can do so by using the `access` option. The following example sets the `access` option to `private`, indicating that all files uploaded should only be accessible to the user account that was used during the upload:

```
aws_s3: {
  myapp: {
    options: {
      accessKeyId: 'AKIAJG27ICB3NTY3SGCQ',
      secretAccessKey: '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',
      region: 'eu-west-1',
      bucket: 'gruntbook.myapp',
      access: 'private'
    },
    files: [{
      src: 'index.html',
      dest: '/'
    }]
  }
}
```

Tip

A list of the possible values for the access option is in the documentation for the AWS `putObject` operation at the following URL (look under the ACL parameter):

http://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWS/S3.html#putObject-property

Enabling concurrent uploads

The default behavior for the `aws_s3` task is to complete uploads, one after another. If we'd like to perform uploads in parallel, we can do so by making use of the `uploadConcurrency` option. In the following example, we indicate that we'd like to have a maximum of `3` files uploaded simultaneously:

```
aws_s3: {
  myapp: {
    options: {
      accessKeyId: 'AKIAJG27ICB3NTY3SGCQ',
      secretAccessKey: 'mT0kL3ANOl88RW+0kP1Sxc89C1DMjp2obv96ubby',
      region: 'eu-west-1',
      bucket: 'gruntbook.myapp',
      uploadConcurrency: 3
    },
    files: [{
      expand: true,
      cwd: 'www',
      src: '**/*',
      dest: '/'
    }]
  }
}
```

Tip

The previous example also demonstrates the configuration involved in recursively uploading the contents of an entire directory.

Deploying over FTP

In this recipe, we will make use of the `ftp-push (0.3.2)` plugin to upload files to a hosting server, using the **File Transfer Protocol** (**FTP**).

FTP has been around since the early days of the Internet and is still in abundant use. As its name implies, it provides a way to transfer files over the Internet and as such, has been the staple for the deployment of resources to web servers since its inception.

Getting ready

In this example, we'll work with the basic project structure that we created in the *Installing Grunt on a project* recipe in Chapter 1, *Getting Started with Grunt*. Be sure to refer to it if you are not yet familiar with its contents.

In addition to the standard project setup, the following recipe will also require an existing user account on the targeted FTP-enabled server. Their credentials are usually provided by the hosting service or the systems administrator in charge of maintaining the server in question.

How to do it...

The following steps take us through creating a simple HTML document and configuring a task that uploads it to a server using FTP:

1. We'll start by installing the package that contains the `ftp-push` plugin, as per the instructions provided in the *Installing a plugin* recipe in Chapter 1, *Getting Started with Grunt*.
2. Next, we'll create a simple HTML document called `index.html` in the project directory, and provide it with the following contents:

   ```
   <html>
     <head>
       <title>MyApp</title>
     </head>
     <body>
       <h1>This is MyApp.</h1>
     </body>
   </html>
   ```

3. Now, we'll add the following `ftp_push` task to our configuration and indicate that we'd like it to upload the `index.html` file to our hosting server:

   ```
   ftp_push: {
     myapp: {
       options: {
         host: '[host]',
         username: '[username]',
         password: '[password]',
         dest: '/www'
       },
       files: [{
         src: 'index.html',
       }]
     }
   }
   ```

   ### Tip

   The `username`, `password`, and `host` options are filled with placeholder values to indicate that you should make use of your own specific access credentials and hosting server address.

   You'll probably want to have the `username` and `password` options stored in a local file, as opposed to a shared repository. Refer to the *Importing external data* recipe of Chapter 1, *Getting Started with Grunt* for an example of how to import configurations from external files. Also, keep in mind that you should exclude files that contain access credentials from your project's code repository.

The `host` option is used to specify the Internet address of the server that you'd like to upload files to. If you are not sure what this value should be, this type of information can usually be provided to you by the hosting provider or the systems administrator in charge of maintaining the server in question.

The `dest` option is required to indicate the destination directory for the files that we'll be uploading. For our example, we'll indicate that files should be uploaded to the `www` directory on the hosting server.

Note that the files configuration for this task supports all the standard Grunt options. You can read more about them at the following URL:

http://gruntjs.com/configuring-tasks#files

4. We can now run the task by using the `grunt ftp_push` command, which should produce output similar to the following:

```
Running "ftp_push:myapp" (ftp_push) task
>> [username] successfully authenticated!
>> /www/index.html transferred successfully.
>> FTP connection closed!
```

Tip

Note that the `ftp_push` task cannot automatically create the destination directory, so the `www` directory needs to exist on the root of the FTP server before running this task.

5. Our `index.html` file should now be present on our hosting server and, if correctly configured, should be accessible via the Internet. Configuring a domain name to point to a site hosted in this fashion is beyond the scope of this book.

Deploying over SFTP

In this recipe, we'll make use of the `sftp` task, which is provided by the `ssh (0.12.2)` plugin to upload files to a hosting server, using the **SSH File Transfer Protocol** (**SFTP**).

The SFTP provides the same functionality as the regular FTP that was discussed in the previous recipe, but with benefits from an added layer of security by making use of the **Secure Shell** (**SSH**) network protocol.

Getting ready

In this example, we'll work with the basic project structure that we created in the *Installing Grunt on a project* recipe in Chapter 1, *Getting Started with Grunt*. Be sure to refer to it if you are not yet familiar with its contents.

In addition to the standard project setup, the following recipe will also require an existing user account on the targeted SSH-enabled server. These credentials are usually provided by the hosting service or the systems administrator in charge of maintaining the targeted server.

How to do it...

The following steps take us through creating a simple HTML document and configuring a task that uploads it to the server using SFTP:

1. We'll start by installing the package that contains the `ssh` plugin, as per the instructions provided in the `Installing a plugin` recipe in Chapter 1.
2. Next, we'll create a simple HTML document called `index.html` in the project directory, and providing it with the following contents:

```
<html>
  <head>
    <title>MyApp</title>
  </head>
  <body>
    <h1>This is MyApp.</h1>
  </body>
</html>
```

3. Now, we'll add the following `sftp` task to our configuration, which indicates that we'd like to have the `index.html` file uploaded to a specific host that supports SFTP:

```
sftp: {
  myapp: {
    options: {
      host: '[host]',
      username: '[username]',
      password: '[password]',
      path: 'www/'
    },
    files: [{
      src: 'index.html',
    }]
  }
}
```

Tip

The `username`, `password`, and `host` options are filled with placeholder values to indicate that you should make use of your own specific access credentials and hosting server address.

You'll probably want to have the `username` and `password` options stored in a local file, as opposed to a shared repository. Refer to the *Importing external data* recipe of Chapter 1, *Getting Started with Grunt*, for an example of how to import configurations from external files. Also, keep in mind that you should exclude files that contain access credentials from your project's code repository.

The `host` option is used to specify the Internet address of the server that you'd like to upload files to. If you are not sure what this value should be, this type of information can usually be provided to you by the hosting provider or the systems administrator in charge of maintaining the server in question.

The `path` option is required to indicate the destination directory for the files that we'll be uploading. For our example, we'll indicate that files should be uploaded to the `www` directory on the hosting server.

Note that the files configuration for this task supports all the standard Grunt options. You can read more about them at the following URL:

http://gruntjs.com/configuring-tasks#files

4. We can now run the task by using the `grunt sftp` command, which should produce output similar to the following:

```
Running "sftp:myapp" (sftp) task
Copied 1 files
```

Tip

Note that the `sftp` task cannot automatically create the destination directory, so the `www` directory needs to exist on the root of the FTP server before running this task.

5. Our `index.html` file should now be present on our hosting server and, if correctly configured, should be accessible via the Internet. Configuring a domain name to point to a site hosted in this fashion is beyond the scope of this book.

There's more...

The `sftp` task provides us with several useful options that can be used in conjunction with its uploading feature. We'll look at how to use a private key and passphrase and using an SSH agent.

Using a private key and passphrase

In case we'd like to make use of a private key and passphrase to access our host server, we can do so by using the `privateKey` and `passphrase` options. In the following example, we will load our private key from the usual location by using the `grunt.file.load` function and provide the passphrase that was used to lock it.

```
sftp: {
  myapp: {
    options: {
      host: '[host]',
      username: '[username]',
      privateKey: grunt.file.read('[path to home]/.ssh/id_rsa'),
      passphrase: '[passphrase]',
      path: 'www/'
    },
    files: [{
      src: 'index.html',
    }]
  }
}
```

Using an SSH agent

If we often make use of SSH to access our servers, we're probably better off making use of an **SSH agent** to store our unencrypted private key after unlocking it the first time during our session. This will allow us to access all the services that make use of our **public key**, without having to enter the passphrase again for the duration of our user session.

Tip

Most Unix-like (this includes OS X) operating systems should have an SSH agent installed and running by default, in which case the following example should work without any initial steps. A Windows user will have to manually install and configure an SSH agent.

The following example makes use of an SSH agent by indicating path to the socket file using the `agent` option:

```
sftp: {
  myapp: {
    options: {
      host: '[host]',
      username: '[username]',
      agent: process.env.SSH_AUTH_SOCK,
      path: 'www/'
    },
    files: [{
      src: 'index.html',
    }]
  }
}
```

Tip

Most SSH agent programs that come packaged with operating systems follow the convention of providing the socket on which it runs by using the `SSH_AUTH_SOCK` environment variable. In our example, we use the standard Node.js `process.env` object to retrieve the value of this environment variable.

Deploying to GitHub Pages

In this recipe, we'll make use of the `gh-pages (0.10.0)` plugin to publish our site to the **GitHub Pages** service.

The GitHub Pages service provides a simple way for GitHub users to host static sites related to themselves, their organizations, or their projects. At the core of this service lies the standard **GitHub** service that provides hosting for **Git**-based code repositories.

Getting ready

In this example, we'll work with the basic project structure that we created in the *Installing Grunt on a project* recipe in Chapter 1, *Getting Started with Grunt*. Be sure to refer to it if you are not yet familiar with its contents.

In addition to the standard project setup, the following recipe will require a GitHub user account that has the appropriate public key (SSH) associated with it.

Tip

Please refer to the following URL for help to create and set up SSH keys for your GitHub account:

https://help.github.com/articles/generating-ssh-keys/

We'll also need to create a repository called `myapp` on GitHub, clone it, and use it as our project folder.

Tip

You can read more about how to create and clone a repository on GitHub at the following URLs:

https://help.github.com/articles/creating-a-new-repository/

https://help.github.com/articles/cloning-a-repository/

How to do it...

The following steps take us through creating a simple HTML document and configuring a task that publishes it to the site of GitHub Pages for our project:

1. We'll start by installing the package that contains the `gh-pages` plugin as per the instructions provided in the *Installing a plugin* recipe in Chapter 1, *Getting Started with Grunt*.
2. Next, we'll create a simple HTML document called `index.html` in the `www` directory, and provide it with the following contents:

   ```
   <html>
     <head>
       <title>MyApp Project Site</title>
     </head>
     <body>
       <h1>Welcome!</h1>
       <h2>This is the MyApp project site.</h1>
     </body>
   </html>
   ```

3. Now, we'll add the following `gh-pages` task to our configuration and indicate that we'd like to have the contents of the `www` directory uploaded to our project's repository:

   ```
   'gh-pages': {
     myapp: {
       options: {
         base: 'www'
       },
       src: '**/*'
     }
   }
   ```

   ### Tip

   The `base` option is used to provide the directory that contains our site. In the case of our example, we'll use the `www` directory, as that is where we created our sample HTML file.

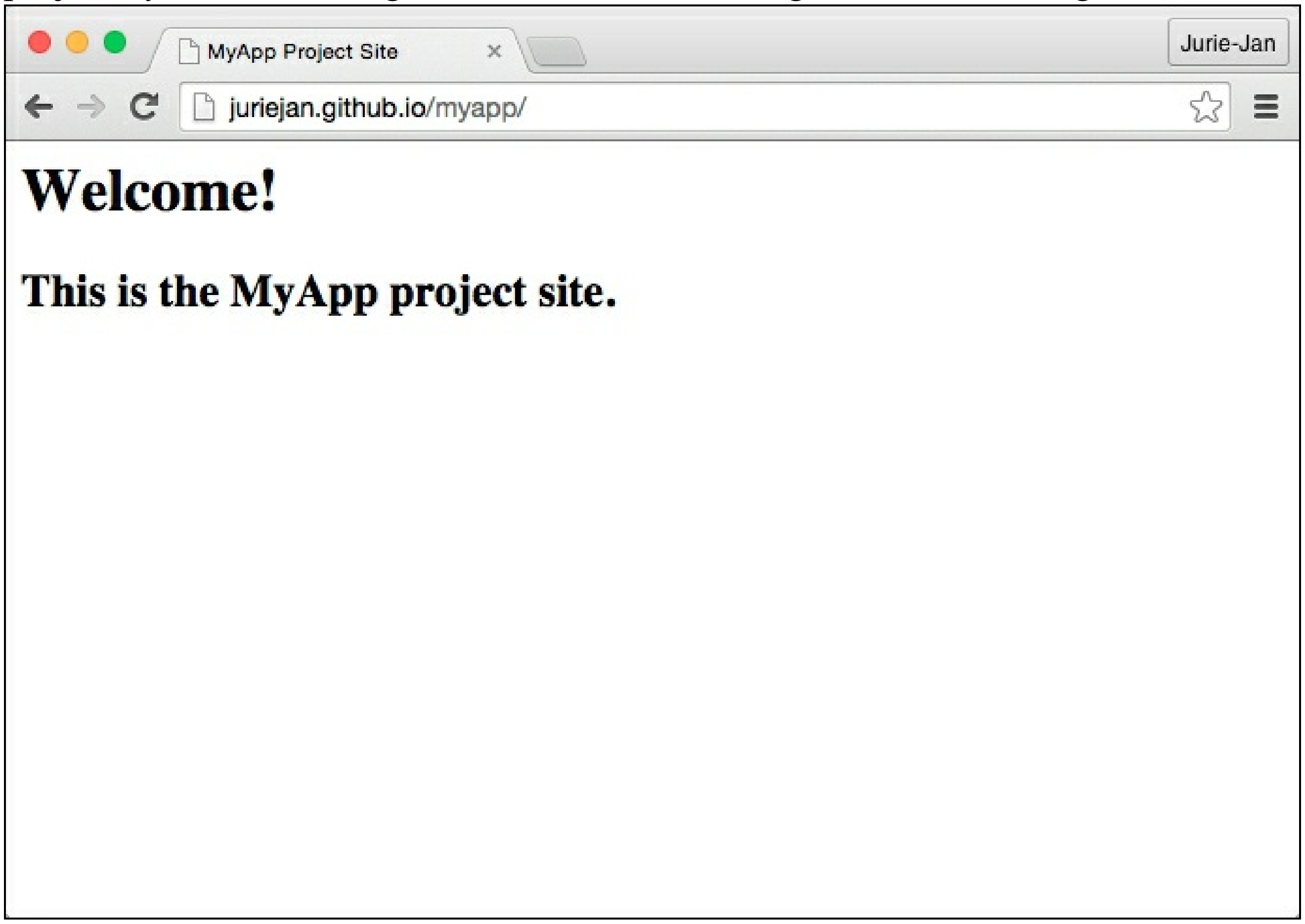
   The `src` option is used to provide files in the base directory which should be transferred to the target repository. For our example, we set it to `**/*` to indicate that we'd like to have all files in the base directory transferred.

4. We can now run the task by using the `grunt gh-pages` command, which should

produce output similar to the following:

```
Running "gh-pages:myapp" (gh-pages) task
Cloning git@github.com:[name]/myapp.git into .grunt/grunt-gh-
pages/gh-pages/myapp
Cleaning
Fetching origin
Checking out origin/gh-pages
Removing files
Copying files
Adding all
Committing
Pushing
```

5. We should now be able to view our project site by navigating to the `[name].github.io/myapp` domain name that was automatically assigned to our project by GitHub. Doing so should look something like the following:



Welcome!

This is the MyApp project site.

Invalidating an AWS CloudFront distribution

In this recipe, we make use of the `invalidate-cloudfront (0.1.6)` plugin to invalidate a AWS CloudFront distribution.

The AWS CloudFront service provides us with an easy way to distribute the files of our websites and applications over a CDN with edge locations all over the world. This leads to faster response times for our intended audience, no matter where they may be in the world.

A side effect of having our file hosted on a CDN is that whenever they are updated at the source, the updates may take a while to reflect at the various edge locations, potentially keeping critical updates from our audience. AWS CloudFront does, however, allow us to indicate that we'd like to refresh the content that is stored on the CDN by invalidating a distribution.

Getting ready

In this example, we'll work with the basic project structure that we created in the *Installing Grunt on a project* recipe in Chapter 1, *Getting Started with Grunt*. Be sure to refer to it if you are not yet familiar with its contents.

In addition to the standard project setup, the following recipe will also require the setup of an AWS user with an AWS access key.

Tip

Refer to the following URL for details of how to obtain your AWS access credentials:

http://docs.aws.amazon.com/general/latest/gr/getting-aws-sec-creds.html

An AWS CloudFront distribution will also need to be created so that we can use it in our recipe. For our example, we'll have it distribute the files contained in the bucket that we created in the *Deploying to AWS S3* recipe, earlier in this chapter.

Tip

Refer to the following URL for more information on how to create an AWS CloudFront distribution:

http://docs.aws.amazon.com/AmazonCloudFront/latest/DeveloperGuide/distribution-web-creating.html

How to do it...

The following steps take us through creating and configuring a task that invalidates a specific AWS CloudFront distribution:

1. We'll start by installing the package that contains the `invalidate-cloudfront` plugin, as per the instructions provided in the *Installing a plugin* recipe in Chapter 1, *Getting Started with Grunt*.
2. Now, we'll add the following `invalidate_cloudfront` task to our configuration and indicate that we'd like to have each of the files that can be found in the `www` directory refreshed on the CloudFront CDN:

   ```
   invalidate_cloudfront: {
     myapp: {
       options: {
         key: '[key]',
         secret: '[secret]',
         distribution: '[distribution id]'
       },
       files: [{
         expand: true,
         cwd: 'www',
         src: '**/*'
       }]
     }
   }
   ```

   ### Tip

   The `key` and `secret` options are filled with placeholder values to indicate that you should make use of your own specific access credentials.

   The `key` option should be set to an **access key ID** that was generated for your user, using the AWS IAM console.

   The `secret` option should be set to the secret access key that was generated for the specific access key ID, you specified in the `key` option. Note that the secret access key is only displayed on creation of the access key ID, so you won't be able to find it in the IAM console if you didn't save it the first time.

   You'll probably want to have the `key` and `secret` options stored in a local file, as opposed to a shared repository. Refer to the *Importing external data* recipe of Chapter 1, *Getting Started with Grunt* for an example of how to import configurations from external files. Also, keep in mind that you should exclude files that contain security credentials from your project's code repository.

3. We can now run the task by using the `grunt invalidate_cloudfront` command, which should produce output similar to the following:

```
Running "invalidate_cloudfront:myapp" (invalidate_cloudfront)
task
Invalidating 1 files: /index.html

0 Completed and 0 In Progress invalidations on: [dist id]

Creating invalidation for 1 files
Invalidation created at https://cloudfront.amazonaws.com/2014-10-
21/distribution/[dist id]/invalidation/[invalidation id]
```

4. The targeted CloudFront distribution should now be in the process of being invalidated. If there are any changes made to the underlying content, it should be reflected at all edge locations on the CloudFront CDN within a few minutes.

Running commands over SSH

In this recipe, we'll make use of the `sshexec` task provided by the `ssh (0.12.2)` plugin to run commands on a remote server over the SSH network protocol.

Running commands on a remote server can become a necessity during the deployment process, and when we run commands, we'd like be sure that it is being done securely. The SSH protocol is the de facto standard to run commands on remote servers, due in part to the improved security that it provides by encrypting all data that is sent over the network.

Getting ready

In this example, we'll work with the basic project structure that we created in the *Installing Grunt on a project* recipe in Chapter 1, *Getting Started with Grunt*. Be sure to refer to it if you are not yet familiar with its contents.

In addition to the standard project setup, the following recipe will also require an existing user account on the targeted SSH-enabled server. These credentials are usually provided by the hosting service or administrator who maintains the targeted server.

How to do it...

The following steps take us through creating and configuring a task that will run a command that prints the date and time on a remote server:

1. We'll start by installing the package that contains the `ssh` plugin, as per the instructions provided in the *Installing a plugin* recipe in Chapter 1, *Getting Started with Grunt*.
2. Now, we'll add the following `sshexec` task to our configuration and set it up to print the date and time on our remote server by running the `date` command:

   ```
   sshexec: {
     date: {
       options: {
         host: '[host]',
         username: '[username]',
         password: '[password]'
       },
       command: 'date'
     }
   }
   ```

   ### Tip

   The `username`, `password`, and `host` options are filled with placeholder values to indicate that you should make use of your own specific access credentials and hosting server.

   You'll probably want to have the `username` and `password` options stored in a local file, as opposed to a shared repository. Refer to the *Importing external data* recipe of Chapter 1, *Getting Started with Grunt* for an example of how to import configurations from external files. Also, keep in mind that you should exclude files that contain access credentials from your project's code repository.

   The `host` option is used to specify the Internet address of the server that you'd like to upload files to. If you are not sure what this value should be, this type of information can usually be provided to you by the hosting provider or the systems administrator in charge of maintaining the server in question.
3. We can now run the task by using the `grunt sshexec` command, which should produce output similar to the following:

   ```
   Running "sshexec:myapp" (sshexec) task
   ```

```
Thu Jan 1 12:00:00 GMT 2015
```

4. If we can see a date and time returned after running the command, we have successfully run a command on the remote server.

There's more...

The `sshexec` task provides us with several useful options that can be used in conjunction with its uploading feature. We'll look at how to use a private key and passphrase and using an SSH agent.

Using a private key and passphrase

In case we'd like to make use of a private key and passphrase to access our host server, we can do so using the `privateKey` and `passphrase` options. In the following example, we will load our private key from the usual location by using the `grunt.file.load` function and provide the passphrase that was used to lock it.

```
sshexec: {
  date: {
    options: {
      host: '[host]',
      username: '[username]',
      privateKey: grunt.file.read('[path to home]/.ssh/id_rsa'),
      passphrase: '[passphrase]'
    },
    command: 'date'
  }
}
```

Using an SSH agent

If we often make use of SSH to access our servers, we're probably better off making use of an SSH agent to store our unencrypted private key after unlocking it the first time during our session. This will allow us to access all the services that make use of our public key, without having to enter the passphrase again for the duration of our user session.

Tip

Most *nix (this includes OS X) operating systems should have an SSH agent installed and running by default, in which case the following example should work without any initial steps. A Windows user will have to manually install and configure an SSH agent.

The following example makes use of an SSH agent by indicating the socket that runs with the `agent` option:

```
sshexec: {
  date: {
    options: {
      host: '[host]',
      username: '[username]',
      agent: process.env.SSH_AUTH_SOCK
    },
    command: 'date'
  }
}
```

Tip

Most SSH agent programs that come packaged with operating systems follow the convention of providing the socket on which it runs, using the `SSH_AUTH_SOCK` environment variable. In our example, we made use of the standard Node.js `process.env` object to retrieve the value of this environment variable.

Chapter 8. Creating Custom Tasks

In this chapter, we will cover the following recipes:

- Creating an alias task
- Creating a basic task
- Accessing project configuration
- Checking for required configurations
- Checking for the successful execution of other tasks
- Running non-blocking code in a task
- Failing a task
- Using command-line parameters
- Enqueuing tasks to run
- Creating a multi-task
- Using options in a task
- Using files in a task

Introduction

To truly begin appreciating the power and flexibility of Grunt, we have to delve into the creation of our own tasks.

On top of the vast array of plugins at our disposal, each of the tasks provided by them can be configured in a variety of ways that should cover most of our requirements. If, however, we do encounter a problem we cannot solve using an existing task and configuration combination, we can easily create a new task to fill in the gap.

We should also view the creation of tasks as the first stepping stone to creating plugins of our own. The methods of creating tasks discussed in this chapter will provide us with a simple laboratory, where we can build tasks that can eventually be transferred directly to a plugin.

Tip

The following two URLs provide a wealth of information on creating tasks and the tools Grunt makes available to build them. It is highly recommended that you go through these links if you are serious about creating tasks:

http://gruntjs.com/creating-tasks

http://gruntjs.com/api/inside-tasks

Creating an alias task

The tasks we configured for our project tend to perform a single function and nothing more. As a project grows, we will probably start to identify groups of tasks that we tend to run together, in a specific order. It is at this point that an alias task can become quite helpful to us as it allows us to group a set of tasks together under a new task name.

Getting ready

In this example, we'll work with the basic project structure we created in the *Installing Grunt on a project* recipe in Chapter 1, *Getting Started with Grunt*. Be sure to refer to it if you are not yet familiar with its contents.

This recipe also contains an example that makes use of the *Setting up a basic web server* and the *Watching files for changes* recipe of Chapter 1, *Getting Started with Grunt*, and the *Rendering Jade templates* recipe in Chapter 3, *Templating Engines* in one setup. Be sure to refer to these if you'd like to gain a deeper understanding of each of these aspects.

How to do it...

The following steps will take us through building a basic website development setup that continuously renders HTML files from templates, serves them using a basic web server, and ensures a clean environment before starting by removing all previously rendered templates:

1. We'll start by installing the packages that contains the `contrib-clean`, `contrib-connect`, `contrib-jade`, and `contrib-watch` plugins as per the instructions provided in the *Installing a plugin* recipe in Chapter 1, *Getting Started with Grunt*.
2. Next, we'll create a simple Jade template called `index.jade` in the `templates` directory, and provide it with the following contents:

   ```
   doctype html
   html
     head
       title Sample Site
     body
       h1 Welcome to my Sample Site!
   ```

3. Now, we can add tasks to watch, render, serve, and clean our configuration:

   ```
   clean: {
     all: ['www']
   },
   jade: {
     www: {
       expand: true,
       cwd: 'templates',
       src: '**/*.jade',
       dest: 'www',
       ext: '.html'
     }
   },
   connect: {
     dev: {
       options: {
         base: 'www'
       }
     }
   },
   watch: {
     www: {
       files: 'templates/**/*.jade',
       tasks: ['jade:www']
   ```

```
    }
  }
```

4. With all our tasks in place, we can now create an alias task called `run`, which ties them all together in the appropriate order. This is done by adding the following to the end of the main function in our Grunt configurations file:

   ```
   grunt.registerTask('run', [
     'clean:all',
     'jade:www',
     'connect:dev',
     'watch:www'
   ]);
   ```

5. We can now run the alias task just like a regular task by using the `grunt run` command, which should produce output similar to the following:

   ```
   Running "clean:all" (clean) task
   >> 0 paths cleaned.

   Running "jade:www" (jade) task
   >> 1 file created.

   Running "connect:dev" (connect) task
   Started connect web server on http://localhost:8000

   Running "watch" task
   Waiting…
   ```

6. As we can see from the output, our configured tasks have all run as per the order specified in the `run` task, and we are now serving the rendered `index.jade` template. To test the setup, we can navigate to `http://localhost:8000` in the browser, which will display the rendered template.

There's more...

There may be situations where we have a specific task that we'd like to run much more than others, so much so that we may come to think of it as the default task. In such a case, we can make use of the `default` task alias. This alias is called when there is no specific task indicated along with the `grunt` command.

We can modify our main recipe to make use of the `default` task by changing the alias task declaration to the following:

```
grunt.registerTask('default', [
  'clean:all',
  'jade:www',
  'connect:dev',
  'watch:www'
]);
```

If we now run the `grunt` command in our project without a task name following it, which should run all the tasks in the same way as it did in the previous section of this recipe.

Creating a basic task

Despite the large array of plugins available to Grunt users, situations might still arise where we would want to create tasks of our own. These situations are well provided for by Grunt, as it provides a set of utilities that make creating new ones quite simple.

Getting ready

In this example, we'll work with the basic project structure we created in the *Installing Grunt on a project* recipe in Chapter 1, *Getting Started with Grunt*. Be sure to refer to it if you are not yet familiar with its contents.

How to do it...

The following steps will take us through creating a simple task that prints the current system date and time.

1. First, we'll register our task with the name `datetime` and provide it with a description and an empty function as a placeholder. The following code shows the entire Grunt file with the task registration code highlighted:

```
module.exports = function (grunt) {
  grunt.initConfig({});
  grunt.registerTask('default', []);
  grunt.registerTask(
    'datetime',
    'Prints out the current date and time.',
    function () {
    }
  );
};
```

2. With the task registered, we can now provide it with the code that will actually print the current date and time. The following code shows the task registration with the newly added code highlighted:

```
grunt.registerTask(
  'datetime',
  'Prints out the current date and time.',
  function () {
    var date = new Date();
    grunt.log.writeln(date.toString());
  }
);
```

Tip

This bit of code makes use of the standard JavaScript `Date` object and Grunt's `grunt.log` utility, which is specifically made available to tasks for logging all types of messages.

You can read more about the `grunt.log` utility at the following URL:

http://gruntjs.com/api/grunt.log

3. We can now try out a newly created task by running the `grunt datetime` command, which should produce output similar to the following:

```
Running "datetime" task
Thu Jan 1 2015 12:00:00 GMT
```

4. If the output of the task has presented us with the current system date and time, we have successfully created and executed our simple custom task.

Accessing project configuration

In order for a task to function appropriately within a project, it often needs a way to get specific details about it. The standard practice for Grunt projects is to supply project-specific details in the configuration object provided in the `grunt.initConfig` function.

Getting ready

In this example, we'll work with the basic project structure we created in the *Installing Grunt on a project* recipe in Chapter 1, *Getting Started with Grunt*. Be sure to refer to it if you are not yet familiar with its contents.

If any of the steps in this recipe seem hard to follow, be sure to check out the *Creating a basic task* recipe provided earlier in this chapter.

How to do it...

The following steps will take us through creating a task that prints out a message determined by the project's configuration.

1. First, we'll add some sample project configuration using the `grunt.initConfig` function. The following code shows the contents of the entire Grunt file with the configuration highlighted:

   ```
   module.exports = function (grunt) {
     grunt.initConfig({
       location: 'Earth',
       datetime: '<%= new Date().toString() %>',
       project: {
         name: 'Life'
       }
     });
     grunt.registerTask('default', []);
   };
   ```

   ### Tip

   Note the use of a template string in the `datetime` configuration. The following URL provides more information on using template strings in configuration values:

   http://gruntjs.com/configuring-tasks#templates

2. Next, we'll register our task with the name `describe` and provide it with a description and an empty function as a placeholder:

   ```
   grunt.registerTask(
     'describe',
     'Describes the situation.',
     function () {
     }
   )
   ```

3. With the task registered, we can fill the empty function with the following code, which retrieves some project configurations and uses it when printing our message:

   ```
   var projectName = grunt.config('project.name');
   var location = grunt.config('location');
   var datetime = grunt.config('datetime');
   grunt.log.write(
     projectName + ' on ' + location + ' at ' + datetime
   );
   ```

Tip

Here, we make use of the `grunt.config` function to fetch data from the configuration object provided in the `grunt.initConfig` function.

Note that dot notation can be used when retrieving the properties from objects in the configuration object, as demonstrated with the `projectName` variable.

Another concept demonstrated with the `datetime` variable is that values retrieved using the `grunt.config` function are rendered, meaning that any string containing templates will be parsed and rendered. If, for some reason, we'd like to fetch the un-rendered value, we can do so by making use of the `grunt.config.getRaw` function.

4. We can now try out our custom task by running the `grunt describe` command, which should produce output similar to the following:

   ```
   Running "describe" task
   Life on Earth at Thu Jan 1 2015 12:00:00 GMT
   ```

5. If the output of the task has presented us with a message containing the values we supplied in our project configuration, we have successfully created and executed a task that makes use of the project configuration.

Checking for required configurations

It's quite common for tasks to require a minimum set of configurations in order for them to function properly. The Grunt framework provides the `grunt.config.requires` function specifically for this case, failing a task if the indicated configuration is not found.

Getting ready

In this example, we'll work with the basic project structure we created in the *Installing Grunt on a project* recipe in Chapter 1, *Getting Started with Grunt*. Be sure to refer to it if you are not yet familiar with its contents.

If any of the steps in this recipe seem hard to follow, be sure to check out the *Creating a basic task* recipe provided earlier in this chapter.

How to do it...

The following steps will take us through creating two tasks: the first will check for the required configurations that we will supply and the second will check for the configurations we will not supply.

1. First, we'll add some sample project configurations using the `grunt.initConfig` function. The following code shows the contents of the entire Grunt file with the configurations highlighted:

   ```
   module.exports = function (grunt) {
     grunt.initConfig({
       integral: 'check',
       complex: {
         important: 'present'
       }
     });
   };
   ```

2. Next, we'll register a task called `complete` that checks for the configurations that we supplied, by adding the following code after our `grunt.initConfig` call:

   ```
   grunt.registerTask(
     'complete',
     'Complete Configuration.',
     function () {
       grunt.config.requires('integral');
       grunt.config.requires('complex.important');
       grunt.log.write('Complete: Success!');
     }
   );
   ```

3. Then, we can register a task called `incomplete`, which checks for a configuration that we have not supplied, by adding the following code:

   ```
   grunt.registerTask(
     'incomplete',
     'Incomplete Configuration.',
     function () {
       grunt.config.requires('missing');
       grunt.log.write('Incomplete: Success!');
     }
   );
   ```

4. With our tasks added, we'll first run the task that will succeed by using the `grunt complete` command, which should produce output similar to the following:

```
Running "complete" task
Complete: Success!
```

5. Then, we can run the task that should fail by using the `grunt incomplete` command, which should produce output similar to the following:

   ```
   Running "incomplete" task
   Verifying property missing exists in config...ERROR
   >> Unable to process task.

   Aborted due to warnings.
   ```

6. As we can see from the output of the last task, it failed due to the `missing` configuration property not being found.

Checking for the successful execution of other tasks

Due to it being detrimental to the reusability of a task, it's not at all common to have tasks depending on one another, due to it being detrimental to the reusability of a task. If tasks are so tightly coupled that they can't run independently, it should strongly be considered to combine them into a single task.

That being said, there are always exceptions, and Grunt provides for this by way of the `grunt.task.requires` function. When called, this function will check whether the task with the specified name has been successfully executed. If it finds that this is not the case, it will fail the current task.

Getting ready

In this example, we'll work with the basic project structure we created in the *Installing Grunt on a project* recipe in Chapter 1, *Getting Started with Grunt*. Be sure to refer to it if you are not yet familiar with its contents.

If any of the steps in this recipe seem hard to follow, be sure to check out the *Creating a basic task* recipe provided earlier in this chapter.

How to do it...

The following steps will take us through creating two tasks. The first task will print a simple message, and the second task will check whether the first one has been successfully executed.

1. First, we'll register a task called `independent` that can run without any tasks having been run, and prints a message when it has done so. This is done by adding the following code to the main function in our Grunt file:

   ```
   grunt.registerTask(
     'independent',
     'Independent task.',
     function () {
       grunt.log.write('Independent: Success!')
     }
   );
   ```

2. Next, we'll register a task called `dependent`, which will fail if the `independent` task did not run successfully. This is done by adding the following code to the main function in our Grunt file:

   ```
   grunt.registerTask(
     'dependent',
     'Dependent task.',
     function () {
       grunt.task.requires('independent');
       grunt.log.write('Dependent: Success!')
     }
   );
   ```

3. To test this setup, we can first run the `dependent` task using the `grunt dependent` command, which should produce output similar to the following:

   ```
   Running "dependent" task
   Warning: Required task "independent" must be run first. Use --
   force to continue.

   Aborted due to warnings.
   ```

4. As we can see from the output, this task has failed due to the `independent` task not having been executed successfully before attempting to run the `dependent` task.
5. Finally, we can run the two tasks we created in succession using the `grunt independent dependent` command, which should produce output similar to the following:

```
Running "independent" task
Independent: Success!
Running "dependent" task
Dependent: Success!
Done, without errors.
```

6. This time around we can see that both the tasks ran without encountering any problems, and printed their respective messages.

Running non-blocking code in a task

The majority of libraries available in the Node.js sphere tend to be implemented in such a way as to be non-blocking. This means that calling a function provided by a library will usually continue to execute the next line of code after being called, even though it has not fulfilled its intended purpose.

These types of functions tend to use either, or both, the event-driven approach or callback functions to indicate their progress. In either of these situations, we will require a function we can call once a task has been completed and for this purpose, Grunt provides us with the `this.async` utility.

Getting ready

In this example, we'll work with the basic project structure we created in the *Installing Grunt on a project* recipe in Chapter 1, *Getting Started with Grunt*. Be sure to refer to it if you are not yet familiar with its contents.

If any of the steps in this recipe seem hard to follow, be sure to check out the *Creating a basic task* recipe provided earlier in this chapter.

How to do it...

The following steps will take us through creating a task that prints the current system time every second for 5 seconds and then exits.

1. First, we'll register a task named `clock` with a description and an empty placeholder function by adding the following code to the end of the main function in our Grunt file:

   ```
   grunt.registerTask(
     'clock',
     'Prints the time every second for 5 seconds.',
     function () {
     }
   );
   ```

2. Next, we'll define some variables inside our placeholder function that we'll be using in our time-printing code:

   ```
   var done = this.async();
   var count = 5;
   var interval = null;
   ```

   ### Tip

   The `done` variable is the focus of this recipe as it demonstrates the usage of the `this.async` function. It is assigned the function that is returned by calling the `this.async` function. This returned function can be called to indicate whether the task has been completed.

   The `count` variable will be used to keep track of the number of times the system time has been printed out. It will be decremented every time we print the date and time, and we'll stop printing the time when it reaches a value of `0`.

   The `interval` variable is used to store the ID of the **timer interval** that is started at the end of the task function. We need this ID to stop the timer interval from running once the `count` variable reaches a value of `0`.

3. With all the required variables in place, we can now define the function that will print the current system date and time, and also stop the timer once it has run its course. The following code should be added immediately after the previous variable definitions:

   ```
   printTime = function () {
   ```

```
  grunt.log.writeln(new Date());
  count -= 1;
  if (count <= 0) {
    clearInterval(interval);
    done();
  }
};
```

Tip

Here, we make use of the `grunt.log` utility to print the current system date and time by providing it with a newly created built-in JavaScript `Date` object, which, by default, will always contain the current date and time.

We also decrement the count variable and check whether it is less than or equal to `zero`. If we find that it is lower than or equal to zero, we make use of the built-in `clearInterval` function to stop the running of the timer interval. We'll be assigning the timer interval ID to the `interval` variable in the next step.

Once we see that the counter has reached `0` and we've stopped the time interval, we can use the `done` function returned to us by the `this.async` function to indicate that the task has finished running.

4. Finally, we'll start a timer interval that runs the function defined in our previous step every `1000` milliseconds by adding the following code immediately after our `printTime` function definition:

   ```
   interval = setInterval(printTime, 1000);
   ```

5. With our task now containing all the necessary code, we can try it out by running the `grunt clock` command, which should produce output similar to the following:

   ```
   Running "clock" task
   Thu Apr 1 2015 12:00:00 GMT
   Thu Apr 1 2015 12:00:01 GMT
   Thu Apr 1 2015 12:00:02 GMT
   Thu Apr 1 2015 12:00:03 GMT
   Thu Apr 1 2015 12:00:04 GMT

   Done, without errors.
   ```

6. As can be seen in the output of the task, it has printed the current system date and time five times, and then stopped.

Failing a task

The success or failure of a task can be used to indicate whether a task was completed without issue, or if a problem was encountered before it could perform its intended function. By default, Grunt will assume that a task has succeeded but it also provides us with a simple way to indicate whether a task has failed and pass on the reason for its failure.

Getting ready

In this example, we'll work with the basic project structure we created in the *Installing Grunt on a project* recipe in Chapter 1, *Getting Started with Grunt*. Be sure to refer to it if you are not yet familiar with its contents.

If any of the steps in this recipe seem hard to follow, be sure to check out the *Accessing project configuration* recipe provided earlier in this chapter.

How to do it...

The following steps will take us through creating a task whose success is based on the value of a project configuration property:

1. We'll start by registering a task called `check` with a description and an empty placeholder function by adding the following code to the end of the main function in our Grunt file:

   ```
   grunt.registerTask(
     'check',
     'Fails if indicated in configuration.',
     function () {
     }
   );
   ```

2. With the task registered, we can now fill the empty function with code that retrieves a project configuration property and uses it to decide whether it should fail the task:

   ```
   var fail = grunt.config('shouldFail');
   if (fail === true) {
     return new Error('Error Message.');
   } else {
     grunt.log.writeln('Success!');
   }
   ```

   ### Tip

   Note that we return a newly created `Error` object containing a message to indicate that the task has failed. If we simply wanted to fail the task without any message, we can just return the `false` value.

3. With our task ready, we can add the required configuration property to indicate that it should not fail. The following code shows the entire `grunt.initConfig` call with the new configuration highlighted:

   ```
   grunt.initConfig({
     shouldFail: false
   });
   ```

4. Now, we can run the task for the first time to see whether it succeeds as expected. This is done by using the `grunt check` command, which should produce output similar to the following:

   ```
   Running "check" task
   ```

```
Success!

Done, without errors.
```

5. Now that we've seen the task succeed, we can change our configuration to indicate that we'd like for it to fail:

   ```
   grunt.initConfig({
     shouldFail: true
   });
   ```

6. If we now run the task again using the `grunt check` command, it should produce output similar to the following:

   ```
   Running "check" task
   Warning: Task "check" failed. Use --force to continue.

   Aborted due to warnings.
   ```

7. As we can see from the output, the task has failed as expected when setting the `shouldFail` configuration property to `true`.

There's more...

Since a task comes in a variety of types and structures, there are a variety of ways in which we'd like to indicate their failure. We'll now look at failing a non-blocking task and aborting a task immediately on failure.

Failing a non-blocking task

In the case that we've made use of the `this.async` function to indicate whether our task should run in a non-blocking way, the return value of the task function becomes meaningless.

If we'd like to fail a non-blocking task, we can do so using the first parameter of the function returned from the call to the `this.async` method. Providing either `false` or an `Error` object using this parameter will indicate that the task has failed.

In the following example, we altered the task from our main recipe to run in a non-blocking way, and fail appropriately:

```
grunt.registerTask(
  'check',
  'Fails if indicated in configuration.',
  function () {
    var done = this.async();
    var fail = grunt.config('shouldFail');
    if (fail === true) {
      done(new Error('Error Message.'));
    } else {
      grunt.log.writeln('Success!');
    }
  }
);
```

Aborting a task immediately on failure

At times, we might want to have a failing task abort the entire Grunt process as soon as a failure is encountered. This type of behavior is usually associated with what is commonly known as a fatal error.

The following example makes use of the `grunt.fail.fatal` utility to indicate that a fatal error has occurred, and that the Grunt process should exit without any further delay:

```
grunt.registerTask(
  'check',
  'Fails if indicated in configuration.',
  function () {
    var fail = grunt.config('shouldFail');
    if (fail === true) {
      grunt.fail.fatal('Error message.');
    } else {
      grunt.log.writeln('Success!');
    }
  }
);
```

Using command-line parameters

In the case that a configurable aspect of a task needs to change quite often, it can be inefficient to add it to the project configuration. This is where the use of command-line parameters can be useful as they provide us with a way to easily provide options to our task, without having to alter our configuration.

Getting ready

In this example, we'll work with the basic project structure we created in the *Installing Grunt on a project* recipe in Chapter 1, *Getting Started with Grunt*. Be sure to refer to it if you are not yet familiar with its contents.

If any of the steps in this recipe seem hard to follow, be sure to check out the *Creating a basic task* recipe provided earlier in this chapter.

How to do it...

The following steps will take us through creating a task that receives two strings using command-line parameters and uses them to print a message:

1. We'll start by registering a task called `welcome` with a description and an empty placeholder function by adding the following code to the end of the main function in our Grunt file:

   ```
   grunt.registerTask(
     'welcome',
     'Displays a welcome message.',
     function () {
     }
   );
   ```

2. With the task registration setup, we can now fill the empty function with code that receives the command-line parameters and uses them to print a message:

   ```
   var message = 'Welcome to ' + city + ', ' + name + '!';
   grunt.log.writeln(message);
   ```

3. Now, we can run the task using some example parameters. This is done by using the `grunt welcome:Aaron:London` command, which should produce output similar to the following:

   ```
   Running "welcome:Aaron:London" (welcome) task
   Welcome to London, Aaron!
   ```

   ### Tip

   Note that if your task makes use of targets, the first parameter will always be assumed to be the name of the target, and will not be passed on to the task function.

4. As we can see from the output, the task has successfully received the parameters we supplied to it and used them to display a welcome message.

Enqueuing tasks to run

There might come a time that we would like to enqueue the running of the existing tasks from within our own task. This practice is not recommended because a well-designed task should usually be focused on a particular function and remain loosely coupled from other tasks. Tasks designed in this fashion generally tend to be more robust and useful in a larger variety of situations.

Getting ready

In this example, we'll work with the basic project structure we created in the *Installing Grunt on a project* recipe in Chapter 1, *Getting Started with Grunt*. Be sure to refer to it if you are not yet familiar with its contents.

If any of the steps in this recipe seem hard to follow, be sure to check out the *Creating a basic task* recipe provided earlier in this chapter.

How to do it...

The following steps will take us through creating two simple tasks that print a string with one of the tasks enqueuing the other inside itself:

1. We'll start by registering a task called `first` that prints out a string. This can be done by adding the following code to the end of the main function in our Grunt file:

   ```
   grunt.registerTask('first', function() {
     grunt.log.write('First Task');
   });
   ```

2. Next, we'll register another task after the one we just added called `second`. This task will also print a string and then proceed to enqueue the `first` task for running:

   ```
   grunt.registerTask('second', function() {
     grunt.log.write('Second Task');
     grunt.task.run('first');
   });
   ```

3. Now that we've created our two tasks, we can try them out by running the `grunt second` command, which should produce output similar to the following:

   ```
   Running "second" task
   Second Task
   Running "first" task
   First Task
   ```

4. As we can see from the output, the `second` task that we ran enqueued the `first` task, which also ran.

Creating a multi-task

Task configurations provide the most common way of tweaking the functionality of the tasks that we use in our project. With configurations, we can set up a task to fulfill our specific needs, and it also allows us to use the same task in various ways within a single project. The different configurations applied to the same task are called **task targets**.

Getting ready

In this example, we'll work with the basic project structure we created in the *Installing Grunt on a project* recipe in Chapter 1, *Getting Started with Grunt*. Be sure to refer to it if you are not yet familiar with its contents.

If any of the steps in this recipe seem hard to follow, be sure to check out the *Creating a basic task* recipe provided earlier in this chapter.

How to do it...

The following steps will take us through creating a simple task that prints the name of the target configuration and the data supplied to it:

1. We'll start by registering a multi-task called `display` using the `registerMultiTask` function by adding the following code to the end of the main function in our Grunt file:

   ```
   grunt.registerMultiTask(
     'display',
     'Displays target name and related configuration',
     function() {
     }
   );
   ```

2. With the task registered, we can fill the empty function with the following code that prints a message containing the name of the specified target and the data provided for it:

   ```
   grunt.log.write(
     'Displaying ' + this.target + ' with ' + this.data
   );
   ```

   ### Tip

   Note that it's only inside the scope of task functions registered using `registerMultiTask` that `this.target` and `this.data` will be populated with the target name and configuration data.

3. Now that we have a registered and functional task, we can set up some sample configuration to our `grunt.initConfig` call, which we'll use to demonstrate the task's behavior:

   ```
   grunt.initConfig({
     display: {
       foo: 'these configurations',
       bar: 'those configurations'
     }
   });
   ```

4. With our task created and configured, we can try it out by running the `grunt display` command, which should produce output similar to the following:

   ```
   Running "display:foo" (display) task
   ```

```
Displaying foo with these configurations
Running "display:bar" (display) task
Displaying bar with those configurations
```

5. We can now also try to indicate that we'd like to use only the `foo` configuration target by running the `grunt display:foo` command, which should produce output similar to the following:

   ```
   Running "display:foo" (display) task
   Displaying foo with these configurations
   ```

Using options in a task

The most common way of tweaking the functionality of a task is by providing options in the project configurations. Options can be specified by providing an object to the `options` property at either the task or target levels. If an option is provided at both the task and target levels, the one provided at the target level will take precedence.

Getting ready

In this example we'll work with the basic project structure we created in the *Installing Grunt on a project* recipe in Chapter 1, *Getting Started with Grunt*. Be sure to refer to it if you are not yet familiar with its contents.

If any of the steps in this recipe seem hard to follow, be sure to check out the *Creating a multi-task* recipe provided earlier in this chapter.

How to do it...

The following steps will take us through creating a multi-task that prints out the value of an option provided at the two available levels in two different tasks:

1. We'll start by registering a multi-task called `display` using the `registerMultiTask` function by adding the following code to the end of the main function in our Grunt file:

   ```
   grunt.registerMultiTask(
     'display',
     "Displays the value of the 'foo' option",
     function() {
     }
   );
   ```

2. With the task registered, we can fill the empty function with the following code that prints a message containing the value of the `foo` option:

   ```
   var options = this.options();
   grunt.log.write("The value of foo is '" + options.foo + "'.");
   ```

   ### Tip

   Note that the `this.options` method provided in the scope of the task function will automatically merge the task and target options, overriding the task options if they are provided in the target.

   For more information about the `this.options` method, visit the following URL:

   http://gruntjs.com/api/inside-tasks#this.options

3. Now that we have a registered and functional task, we can set up some sample configuration, which we'll use to demonstrate the task's behavior:

   ```
   grunt.initConfig({
     display: {
       options: {
         foo: 'initial'
       },
       first: {},
       second: {
         options: {
           foo: 'override'
         }
   ```

```
      }
    }
  });
```

4. With our task created and supplied with a sample configuration, we can try it out with the `first` target by running the `grunt display:first` command, which should provide output similar to the following:

   ```
   Running "display:first" (display) task
   The value of foo is 'initial'.
   ```

5. Now that we've seen how the `first` target behaves, we can try it out with the `second` target by running the `grunt display:second` command, which should produce output similar to the following:

   ```
   Running "display:second" (display) task
   The value of foo is 'override'.
   ```

Using files in a task

Tasks are commonly required to access a specific set of files, either using the data they contain, or altering it in some way. Grunt provides a uniform way of specifying sets of files using the `src`, `dest`, and `files` properties that can be used at either the task or target levels.

Getting ready

In this example, we'll be working with the basic project structure we created in the *Installing Grunt on a project* recipe in Chapter 1, *Getting Started with Grunt*. Be sure to refer to it if you are not yet familiar with its contents.

If any of the steps in this recipe seem hard to follow, be sure to check out the *Creating a multi-task* recipe provided earlier in this chapter.

How to do it...

The following steps will take us through creating some sample data files and a simple task that prints their contents:

1. Let's start by creating two sample files called `one.dat` and `two.dat` in a new directory called `data` and providing them with the following contents:

   ```
   one.dat - Initial sample data.
   two.dat - Some more sample data.
   ```

2. Next, we'll register a multi-task called `display` using the `registerMultiTask` function by adding the following code to the end of the main function in our Grunt file:

   ```
   grunt.registerMultiTask(
     'display',
     "Displays the contents of the specified files.",
     function() {
     }
   );
   ```

3. With the task registered, we can now fill the empty function with the following code that iterates over the indicated files and prints out their contents:

   ```
   this.files.forEach(function(file) {
     file.src.forEach(function(filepath) {
       var content = grunt.file.read(filepath);
       grunt.log.write(content);
     });
   });
   ```

   ### Tip

   Note that the `this.files` property is only provided inside the scope of a multi-task's function. It provides a normalized list of all file configurations provided at both the task and target levels.

   For more information about the `this.files` property, visit the following URL:

   http://gruntjs.com/api/inside-tasks#this.files

4. In order to demonstrate our task, we can now add some configuration to our `grunt.initConfig` call, which will target all the files contained in the `data` directory:

```
grunt.initConfig({
  display: {
    sample: {
      src: 'data/*'
    }
  }
});
```

5. With everything set up, we can now give the task a try by running the `grunt display` command, which should produce output similar to the following:

```
Running "display:sample" (display) task
Initial sample data.
Some more sample data.
```

There's more...

Apart from just reading from files, we will probably reach a point where we'd like to write to a file. The following steps alter the previous recipe in such a way as to write the contents of the indicated files to a destination file instead of printing it to the console:

1. First, we'll change the code in the function of the `display` task we registered earlier to the following:

   ```
   this.files.forEach(function(file) {
     var content = file.src.map(function(filepath) {
       return grunt.file.read(filepath);
     }).join('');
     grunt.file.write(file.dest, content);
   });
   ```

2. Next, we'll alter the project configuration to indicate a destination file that the output should be written to:

   ```
   grunt.initConfig({
     display: {
       sample: {
         src: 'data/*',
         dest: 'output.dat'
       }
     }
   });
   ```

3. We can now try out our modified task by running the `grunt display` command again, which should produce output similar to the following:

   ```
   Running "display:sample" (display) task
   ```

4. A file named `output.dat` with the following contents should now be present in our project folder:

   ```
   Initial sample data.
   Some more sample data.
   ```

Chapter 9. Authoring Plugins

In this chapter, we will cover the following recipes:

- Finding plugins
- Contributing to a plugin
- Setting up a basic plugin project
- Creating a plugin task
- Writing tests for a plugin task
- Adding documentation for a plugin
- Publishing a plugin

Introduction

For the most part, Grunt is a framework that allows developers to package and configure the various tools available to web application developers in a uniform fashion. In the Grunt sphere, tools are packaged into what are called plugins, and these plugin packages all conform to Grunt operation and configuration conventions.

Some of the more popular plugins are provided by the Grunt core team, but all the others are contributed by the community of developers that use Grunt in their projects. Now that we're making use of Grunt in our project, we are part of this community, and we can take it upon ourselves to assist in the creation and maintenance of these plugins.

Finding plugins

Before we even consider creating a plugin of our own, we should first determine whether there isn't already a plugin out there that will satisfy our requirements. In most cases, you will find that someone else has already ventured to solve the problems you are faced with now, and has done most if not all the work for you.

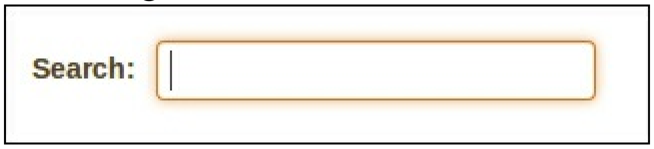
Getting ready

The most obvious place to start looking for a plugin would be on the Internet. So the first thing we'll need to do is open our favorite web browser and get ready to navigate.

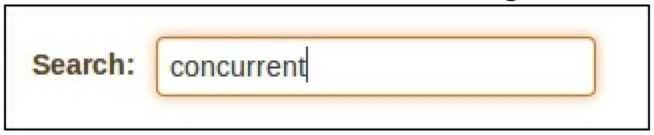
How to do it...

The following steps will take us through navigating to the Grunt project's website and using it to search for a plugin that we can use to run tasks concurrently.

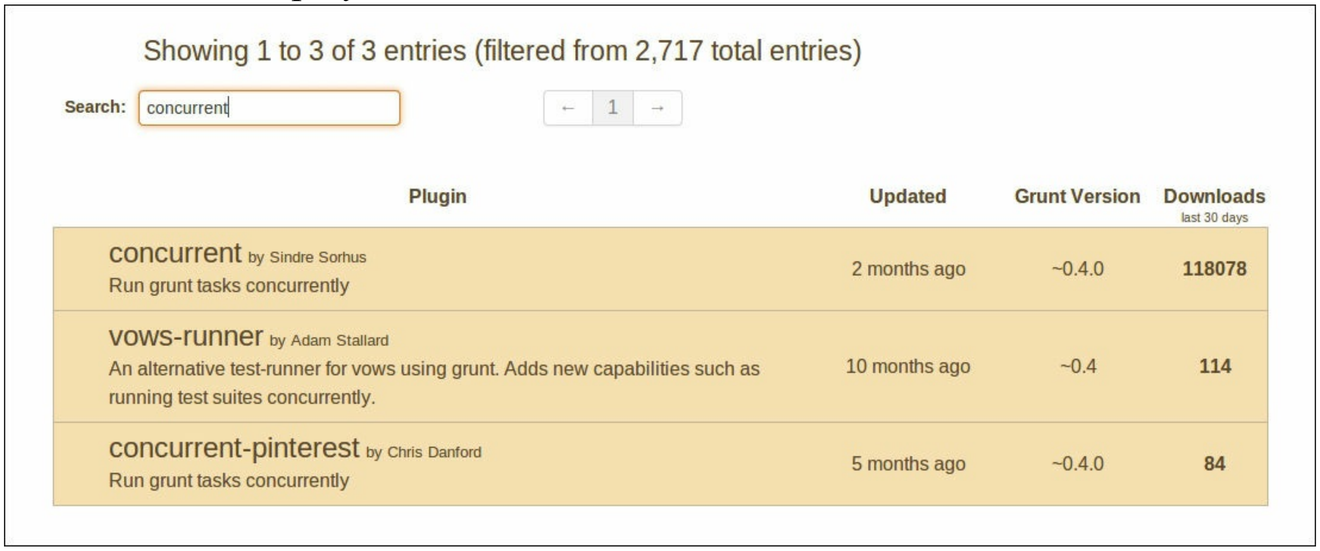
1. First, we'll navigate to the official Grunt plugin listing page by entering the following URL in our web browser: http://gruntjs.com/plugins.
2. Next, we'll select the search input textbox that should look something like the following:



3. With the textbox selected, we can now proceed to type the term we wish to search for; in this case we'll be entering `concurrent` as our search term:



4. Once we've finished typing, we should see the list automatically loading for a little while and then display the search results:



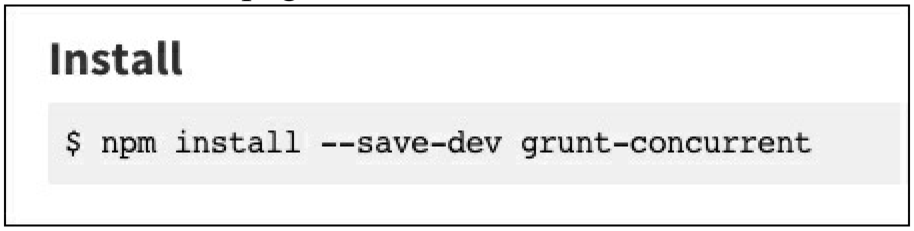
Showing 1 to 3 of 3 entries (filtered from 2,717 total entries)

Search: concurrent

Plugin	Updated	Grunt Version	Downloads last 30 days
concurrent by Sindre Sorhus Run grunt tasks concurrently	2 months ago	~0.4.0	118078
vows-runner by Adam Stallard An alternative test-runner for vows using grunt. Adds new capabilities such as running test suites concurrently.	10 months ago	~0.4	114
concurrent-pinterest by Chris Danford Run grunt tasks concurrently	5 months ago	~0.4.0	84

5. Now that we see a plugin listed that looks like it might be what we're looking for,

we can click on the list item to review its documentation. The installation instructions for the plugin should usually be somewhere near the top of the documentation page.



Install

```
$ npm install --save-dev grunt-concurrent
```

Contributing to a plugin

Once you've found a plugin that closely matches your needs, you might find that some aspect of it is either broken, incomplete, or missing. This is where you can step in and contribute to the project in a variety of ways.

Contributing to a plugin project provides you with the advantage of getting exactly what you want from it, without having to create the entire project yourself. It's also beneficial to others, such as yourself, who have encountered the issue you face or require the same additional features.

Getting started

1. A **GitHub** account is required to make any of the contributions mentioned in this recipe. If you don't already have an account, creating one is as simple as visiting GitHub's home page and filling out the registration form there at https://github.com/.

   ### Tip

   The following URL provides another good all-round introduction to Git and GitHub:

   https://guides.github.com/activities/hello-world/

2. Familiarize yourself with the practice of creating **issues** in the GitHub Issues section. The mechanism to create and manage issues is quite simple, but it's very important to apply good practices when making use of it. Well-written issues make all the difference and will help you get what you need in a shorter period of time. You can find out more about good issue-writing practices at the following location:

   http://wiredcraft.com/posts/2014/01/08/how-we-write-our-github-issues.html
3. In order to retrieve the code and documentation for any of the plugin projects, you will need to have the Git version control software installed. You can find out more about the various installation options for Git at the following URL:

   http://git-scm.com/book/en/v1/Getting-Started-Installing-Git
4. If you'd like to contribute the changes to code or documentation that you've made to a project hosted on GitHub, you will have to familiarize yourself with the pull request workflow. More information on the pull request workflow can be found at the following URL:

   https://help.github.com/articles/using-pull-requests/
5. To get the best out of your contribution efforts to the Grunt project and its plugins, be sure to familiarize yourself with the Grunt contribution guide, which can be found at the following URL:

   http://gruntjs.com/contributing
6. Be sure to carefully study the documentation of the project that 're considering contributing to ensure that you are using it correctly, and that the feature that you're

looking for is not already available. The documentation of a Grunt plugin project should always be found in the `README.md` file in the root of the project's repository.

Tip

GitHub will display the `README.md` document located in the repository root as the repository's home page by default.

7. Each project might also have its own specific contribution guidelines that need to be followed. These are usually found either at the end of `README.md` or in the `CONTRIBUTING.md` document at the root of the project's repository.

How to do it...

There are many ways to contribute to an existing plugin. Let's list them according to their difficulty, starting with the simplest one:

1. Helping with an existing issue by commenting on it's contents, reproducing it's error, or providing a resolution to the stated problem.
2. Adding issues for faults as you encounter them. Keep in mind that a new issue is only helpful if you clearly state the context and problem. What's even more helpful is to provide a potential solution to the problem if you can think of one.

   Tip

   Be sure to check for existing issues with a similar subject. If the issue focuses on the same subject, it is usually better to just add to it. However, if it slightly differs from the existing one, it might be a good idea to refer to it in a new issue.

   It's also recommended to review the documentation of a plugin before submitting an issue. This is to ensure that the issue or behavior you experience is not expected and that you are actually using the plugin as intended by its authors.

3. Adding issues for possible improvements. As always, it's important to clearly state the context and concept of the improvement.

   Tip

   Be sure to check issues and pull requests to see whether the feature has not been requested or developed yet. Also, review the documentation to ensure that it is not yet available in some other form.

4. Submitting updates and improvements to the documentation of the plugin. This requires forking the plugin project and submitting a pull request that contains the changes.
5. Resolving issues and submitting the changes to the code base. This requires forking the project and submitting a pull request. If the reported issue is not breaking tests yet, it's essential to add a test for the specific use case that was causing the error.
6. Implementing new features and submitting the changes to the code base. This requires forking the plugin project and submitting a pull request. Keep in mind that adding a feature requires the documentation for the project to be updated and tests to be created that ensure that the new feature behaves as documented.

Setting up a basic plugin project

At the base of every Grunt plugin lies a Node.js project that contains information about its purpose, version, dependencies, and so forth. Due to the basic project structure for all Grunt plugins being pretty much the same, we'll make use of a project generator to provide us with a starting point.

In this recipe, we'll make use of the **Yeoman** project's scaffolding tool to generate our basic Grunt plugin project. It provides generators for a large variety of project setups, all of which have Grunt as their core automation tool.

Tip

You can learn more about the Yeoman project at the following URL:

http://yeoman.io/

Getting started

The only requirement for this recipe is a global installation of Node.js, with Grunt installed into it as per the *Installing the Grunt CLI* recipe in Chapter 1, *Getting Started with Grunt*. Be sure to refer to it if you are not yet familiar with its contents.

How to do it...

The following steps take us through installing the Yeoman tool and using it to generate a basic Grunt plugin project.

1. First, we'll need to install the Yeoman tool globally using the **npm** utility. This is done by entering the following command:

   ```
   $ npm install --global yo
   ```

2. We'll also need to install the Yeoman generator that is specifically geared toward generating a Grunt plugin project. This is done by entering the following command:

   ```
   $ npm install --global generator-gruntplugin
   ```

3. Next, we'll create a directory called `grunt-myplugin`, which will contain our plugin project, and navigate to it using a command similar to the following:

   ```
   $ mkdir grunt-myplugin && cd grunt-myplugin
   ```

4. With Yeoman and the required generator installed, we can now use it to generate our plugin project by running the following command:

   ```
   $ yo gruntplugin
   ```

5. After running the previous command, you will be presented with a series of questions that will assist the generator in creating the project. This interaction should look something like the following:

   ```
   ? Plugin Name: grunt-myplugin
   ? Description: The best Grunt plugin ever.
   ? Version: 0.0.1
   ? Project git repository: git://github.com/me/grunt-myplugin
   ? Project homepage: http://github.com/me/grunt-myplugin
   ? License: MIT
   ? Author name: Me
   ? Author email: me@example.com
   ? Author url: http://me.example.com/
   ? What versions of node does it run on? >= 0.8.0
   ? What version of grunt does it need? ~0.4.2
   ```

   ### Tip

   All the references to 'me' in the previous example are where you should enter your own name, account name, and credentials.

6. After all the questions have been answered, the generator will create all the files required for a simple plugin with a single functioning task. The output, which is similar to the following, will inform us of these actions:

   ```
      create tasks/myplugin.js
      create test/expected/custom_options
      create test/expected/default_options
      create test/fixtures/123
      create test/fixtures/testing
      create test/myplugin_test.js
      create .jshintrc
      create .gitignore
      create .editorconfig
      create README.md
      create Gruntfile.js
   ```

7. With all the necessary files created, we now need to install the dependencies required for the project to function. These dependencies are listed in the generated `package.json` file and can be installed using the following command:

   ```
   $ npm install
   ```

   Tip

   This command uses the `package.json` file, which is found in the current directory, so be sure to have navigated to the `grunt-myplugin` directory we created earlier, if you have not done so already.

8. The best way to confirm that the project has been successfully set up (apart from actually using it in a Grunt project) would be to run the generated tests. This can be done using the following command:

   ```
   $ npm test
   ```

9. Successfully running the tests should produce output similar to the following:

   ```
   > grunt-myplugin@0.0.1 test /home/me/projects/grunt-myplugin
   > grunt test

   Running "clean:tests" (clean) task

   Running "myplugin:default_options" (myplugin) task
   File "tmp/default_options" created.

   Running "myplugin:custom_options" (myplugin) task
   File "tmp/custom_options" created.
   ```

```
Running "nodeunit:tests" (nodeunit) task
Testing myplugin_test.js..OK
>> 2 assertions passed (18ms)
```

Creating a plugin task

The functionality of Grunt plugins are mostly contained inside the tasks they provide. The plugin project scaffolding provided by the Yeoman tool creates one such task for us to work from or use as a reference when creating our own.

Getting started

In this recipe, we'll work with the basic project structure we created in the *Setting up a basic plugin project* recipe earlier in this chapter. Be sure to refer to it if you are not yet familiar with its contents.

This recipe also contains concepts that are introduced in the *Creating a multi task*, *Using options in a task* and *Using files in a task* recipes that can be found at the end of Chapter 8, *Creating Custom Tasks*. Be sure to refer to these recipes if you'd like to gain a deeper understanding of the concepts they introduced.

How to do it...

The following steps take us through creating a task that concatenates all the indicated source files, and then prepends a comment containing a timestamp and location to the result.

1. We'll start by creating a new file called `timestamp.js` in the `tasks` directory that will contain our task code. It's good practice to have the file that contains a task named after the task itself.
2. Next, we'll set up the code module that will contain our task code and register the new task inside it by filling the new file with the following code:

```
module.exports = function (grunt) {
  grunt.registerMultiTask(
    'timestamp',
    'Perpends a files contents with a timestamp.',
    function () {
    }
  );
};
```

   ### Tip

   The `module.exports` object is automatically made available by Node.js for each file. Whatever gets assigned to it is what will be made available when the file is imported into another file. You can read more about `module.exports` at the following URL:

   https://nodejs.org/api/modules.html#modules_module_exports

3. The first thing we'll need to do inside our task is retrieve the options for the task target and store it in the `options` variable. We'll also provide some default values for the available options to be sure that our task works even without any options being provided. The following code takes care of this and can be added at the top of the task function:

```
var options = this.options({
  datetime: new Date(0),
  location: 'London'
});
```

   ### Tip

   The default for the `datetime` option is the very earliest available value in the Unix time range. This value is determined by creating a standard JavaScript `Date` object and providing `0` as its only parameter.

4. The next thing we'll do is create a `comment` variable that contains an empty string, and then add the contents of our comment to it piece by piece. The following code does this for us:

```
var comment = '';
comment += '// ' + options.datetime.toGMTString();
comment += ' at ' + options.location + '\n';
```

5. With our comment string ready, we can now concatenate all the source files, and write the comment and result to the indicated destination file. The following code does this for all files indicated in the task configuration:

```
this.files.forEach(function(file) {
  var src = file.src.map(function (path) {
    return grunt.file.read(path);
  }).join('');
  grunt.file.write(file.dest, comment + src);
});
```

6. Now that we've got our task registered and functional, we can add some configurations that we can test it with. Let's add two targets: one called `default_options` that tests the task without any options provided and another called `custom_options` that tests the task with all the possible options provided. We can do this by adding the following configuration in our Gruntfile:

```
timestamp: {
  default_options: {
    options: {}
  },
  custom_options: {
    options: {
      datetime: new Date(Date.UTC(2014, 0, 1)),
      location: 'New York'
    }
  }
}
```

Tip

We're making use of the `Date.UTC` function here to ensure that the behavior of this code doesn't differ between the various time zones. Setting the date in this way will always amount to a date and time value in the GMT time zone.

7. In order for these task targets to work, they will also have to indicate the source files they would like to read from and the destination file they would like the result to be written to. For our test, we'll use the files that were created by the project generator in the `test/fixtures` directory as sources, and write the result to the `tmp/timestamp` directory. This can be done by adding the following `files` configurations:

```
default_options: {
  options: {},
  files: {
    'tmp/timestamp/default_options': [
      'test/fixtures/testing',
      'test/fixtures/123'
    ]
  }
},
custom_options: {
  options: {
    datetime: new Date(Date.UTC(2014, 0, 1)),
    location: 'New York'
  },
  files: {
    'tmp/timestamp/custom_options': [
      'test/fixtures/testing',
      'test/fixtures/123'
    ]
  }
}
```

8. Finally, we can run our task using the `grunt timestamp` command, which should produce output similar to the following:

```
Running "timestamp:default_options" (timestamp) task
Running "timestamp:custom_options" (timestamp) task
```

9. To confirm that the tasks ran correctly, we can check the contents of the files created in the `tmp/timestamp` directory. The `custom_options` file, for example, should have content similar to the following:

```
// Wed Jan 01 2014 00:00:00 GMT (SAST) at New York
Testing1 2 3
```

Writing tests for a plugin task

The creation of tests forms an essential part of the development of all programming modules, which includes Grunt plugins. Tests provide a way for us to confirm that our tasks work and keep working as expected in a variety of situations.

Getting started

In this recipe, we'll continue to work on the project we created in the *Creating a plugin task* recipe earlier in this chapter. Be sure to refer to it if you are not yet familiar with its contents.

How to do it...

The following steps take us through creating expected output files and a test suite that checks that the actual output files from the task match them.

1. Before we get started on creating the actual test suite, we'll create a file called `default_options` in the `test/expected/timestamp` directory that contains the expected output of running the task with the `default_options` target:

   ```
   // Thu, 01 Jan 1970 00:00:00 GMT at London
   Testing1 2 3
   ```

2. Next, we'll create another file called `custom_options` in the same directory that contains the expected output of running the file with the `custom_options` target:

   ```
   // Wed, 01 Jan 2014 00:00:00 GMT at New York
   Testing1 2 3
   ```

3. Now that we've got the files containing the expected output ready, we can begin to set up the test suite that will actually compare their contents with the task's results. We'll start by creating a file called `timestamp_test.js` in the `test` directory and fill it with the following code:

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

   module.exports.timestamp = {
   };
   ```

   ### Tip

   The Grunt library is imported and assigned to the `grunt` variable at the beginning of the file for later use in the tests.

   We also assign an empty object to a property named after the task in the `module.exports` object. We'll add our test functions to this object.

4. With the basic framework for our test suite in place, we can now add a test for each of our expected results. We'll name the tests after the targets that they'll be testing, just like the expected result files we created earlier. The tests are added by adding the following properties to the exported object in the `timestamp_test.js` file:

   ```
   module.exports.timestamp = {
     default_options: function (test) {
       test.expect(1);
   ```

```
    var actual = grunt.file.read(
      'tmp/timestamp/default_options'
    );
    var expected = grunt.file.read(
      'test/expected/timestamp/default_options'
    );
    test.equal(
      actual,
      expected,
      'should work without specifying options.'
    );
    test.done();
  },
  custom_options: function (test) {
    test.expect(1);
    var actual = grunt.file.read(
      'tmp/timestamp/custom_options'
    );
    var expected = grunt.file.read(
      'test/expected/timestamp/custom_options'
    );
    test.equal(
      actual,
      expected,
      'should work with custom options.'
    );
    test.done();
  }
};
```

Tip

The format of tests is usually determined by the framework that runs them. In the case of our example, which is based on the generated Grunt plugin project, we use the **nodeunit** framework. You can read more about the framework at the following URL:

https://github.com/caolan/nodeunit

5. Now that we have our test suite set up, we need to indicate to the `test` alias task that it should run the timestamp task before running the tests. This causes the task to produce the resulting files that can then be tested against files that have been set up as the expected results. This is done by adding the `timestamp` task to the `test` alias task before the `nodeunit` task that runs the tests:

```
grunt.registerTask('test', [
  'clean',
  'myplugin',
  'timestamp',
  'nodeunit'
]);
```

6. Finally, we can run the tests using either the `npm test` or `grunt test` command, which should produce output similar to the following:

```
Running "clean:tests" (clean) task
Cleaning tmp...OK

Running "myplugin:default_options" (myplugin) task
File "tmp/default_options" created.

Running "myplugin:custom_options" (myplugin) task
File "tmp/custom_options" created.

Running "timestamp:default_options" (timestamp) task

Running "timestamp:custom_options" (timestamp) task

Running "nodeunit:tests" (nodeunit) task
Testing myplugin_test.js..OK
Testing timestamp_test.js..OK
>> 4 assertions passed (73ms)
```

7. We can now see by the output near the end that our tests have run successfully and we can keep running them during development to make sure we haven't broken anything with our latest changes.

Adding documentation for a plugin

High-quality documentation is essential for the success of most software development projects. In the context of a Grunt plugin, its main purpose is to provide instructions and information on the usage of the plugin and its tasks. As with most projects hosted on GitHub, the documentation for a plugin is located in the `README.md` file, and written in the **Markdown** format.

Tip

For more information on the general Markdown format, you can visit the following URLs:

http://daringfireball.net/projects/markdown/

https://help.github.com/articles/github-flavored-markdown/

Getting started

In this recipe, we'll continue to work on the project we created in the *Creating a plugin task* recipe earlier in this chapter. Be sure to refer to it if you are not yet familiar with its contents.

How to do it...

The following steps take us through reviewing and adding to the documentation generated by Yeoman and the `gruntplugin` generator we used earlier.

1. Let's start by using our favorite editor to open the `README.md` file located in our project root.
2. First, we'll check the title and description right at the beginning of the documentation file. As per our initial generation of the project, the title `grunt-myplugin` is probably still fine for us. We do, however, have a new task in this plugin that we should probably mention in the description. Let's change it to the following:

   ```
   > Our very first plugin with our very first "timestamp" task.
   ```

3. Next, we can review the `Getting Started` section to ensure that it still suits our project. In this case, the generated instructions should still be sufficient.
4. Going further along in the documentation, we should now see the tasks contained in the plugin being listed. The first and only task currently listed is the `myplugin` task that we generated along with the project. Let's begin writing the documentation for our `timestamp` task by adding the following header after the `myplugin` tasks section:

   ```
   ## The "timestamp" task
   ```

5. The first part we'll add to our new task's section will be the overview. This should provide a basic explanation of the goal and operation of the task. The following should be sufficient for our new task:

   ```
   ### Overview
   The `timestamp` task can be used to concatenate files and prepend
   the result with the a comment containing the provided time and
   location.
   ```

6. Now we should probably provide instructions with regard to the options that are available to tweak the task. Let's add the following to explain the `datetime` and `location` options:

   ```
   ### Options

   #### options.datetime
   Type: 'Date'
   Default value: Minimum Unix time value
   ```

````
A Date object that contains the date and time you'd like to print
out in the prepended comment.

#### options.location
Type: 'String'
Default: 'London'

A string value that contains the location that you'd like to
print out in the prepended comment.
````

7. Next, we'll add some usage examples for our task. The following examples illustrate using task with the default options, and with all the available options set:

~~~
### Usage Examples

#### Default Options
In this example we make use of all the default values of the
options. This should concatenate the provided files and prepend
the result with a comment containing the minimum Unix time, along
with the location as 'Earth'.

```js
grunt.initConfig({
  timestamp: {
    options: {},
    files: {
      'dest/default_options': [
        'src/foo',
        'src/bar'
      ]
    },
  },
})
```

#### Custom Options
In this example we set all possible options. The `datetime`
options is set to the first day in 2014 and the location is set
to 'New York'.

```js
grunt.initConfig({
  timestamp: {
    options: {
      datetime: new Date(2014, 0, 1),
      location: 'New York'
    },
    files: {
      'dest/default_options': [
~~~

~~~
        'src/foo',
        'src/bar'
      ]
    },
  },
})
```

This should produce a comment of the following form:
```js
// Wed, 01 Jan 2014 00:00:00 GMT at New York
```
~~~

8. Finally, we can review the contents of the `Contributing`, `Release History` and `License` sections to ensure there isn't anything we'd like to change in, or add to, our project.

Publishing a plugin

Having a plugin published could be your introduction to the world of open source development and collaboration. As a project gains traction in the community and becomes useful to others, you may begin to see contributions being made to the project; contributions such as the fixing of issues and implementing of features that you may or may not have had on your roadmap.

Getting started

In this recipe, we'll publish the project that we've been creating throughout this chapter, culminating with the project from the *Adding documentation for a plugin* recipe. Be sure to refer to it and its preceding recipes if you'd like to see how we got to this point.

Before we can publish anything to the **node package manager** registry, we'll need to register as a user with the package hosting service. This can be done at https://www.npmjs.com/signup.

How to do it...

The following steps take us through publishing our plugin project to the node package manager registry:

1. To start with, we'll need to identify ourselves to the NPM service by entering the `npm adduser` command in the command line and entering the user credentials we registered with at the www.npmjs.com website.
2. Next, we should make sure that we are in the root directory of the plugin project we wish to publish. If we are not yet there, we should navigate to it before we continue.
3. Once we're in the appropriate directory, publishing our plugin is as simple as running the `npm publish` command, which should produce output similar to the following:

   ```
   + grunt-myplugin@0.0.1
   ```

4. For the sake of this example, we'll also remove our plugin from the registry using the `npm unpublish --force` command, which should produce output similar to the following:

   ```
   npm WARN using --force I sure hope you know what you are doing.
   - grunt-myplugin@0.0.1
   ```

Chapter 10. Static Sites

In this chapter, we will cover the following recipes:

- Setting up a basic site project
- Adding a page to the site
- Adding content to the site
- Adding data to the site
- Creating and using a site layout
- Generating pages from a collection
- Creating a template helper
- Using a plugin

Introduction

One of the areas where Grunt really shines is in the management of static website projects. Using Grunt for generating a static website can result in a fast and easily modifiable site, replacing the complexity and cost of building a traditional server-generated site when its advantages would not be useful to the situation.

In this chapter we'll be focusing on making use of the **Assemble** framework to build a static website. At its core, Assemble is a Grunt plugin, but it also uses Grunt to bring together all its other related tools.

As an example, we'll build a simple static website to represent us on the Internet in an individual capacity. Think of it as a résumé made available to the public via the **World Wide Web** (**WWW**).

Setting up a basic site project

To speed up the process of starting a new project based on the **Assemble** framework, we'll make use of a **Yeoman** generator provided by the developer community. The generator will produce a minimal project setup, with everything we'll need to get started.

Getting started

The only requirement for this recipe is a global installation of Node.js, with Grunt installed on top of it as per the *Installing the Grunt CLI* recipe in Chapter 1, *Getting Started with Grunt*. Be sure to refer to it if you are not yet familiar with its contents.

How to do it...

The following steps take us through installing the Yeoman tool and using it to generate a basic Assemble project.

1. First we'll need to install the Yeoman tool globally using the `npm` utility. This is done by entering the following command:

   ```
   $ npm install --global yo
   ```

2. We'll also need to install the Yeoman generator that is specifically geared toward generating an Assemble project. This is done by entering the following command:

   ```
   $ npm install --global generator-assemble
   ```

3. Next we'll create a directory called `mysite` that will contain our static website project, and navigate into it using a command similar to the following:

   ```
   $ mkdir mysite && cd mysite
   ```

4. With Yeoman and the required generator installed, we can now use it to generate our plugin project by running the following command:

   ```
   $ yo assemble
   ```

5. After running the previous command, we will be presented with a series of questions that will assist the generator in creating the project. This interaction should look something like the following:

   ```
   ? Your project name: mysite
   ? Your project description: The best project ever.
   ? Would you mind telling me your username on Github? me
   ? Do you want to install Ass
   emble plugins? No
   ```

   ### Tip

   Be sure you replace the example `me` GitHub username with your own, and to opt-out of installing any extra Assemble plugins. Making use of extra plugins will be discussed later in this chapter.

6. After the questions have been answered, the generator will create the files required for a simple static website project. Output similar to the following will inform us of these actions:

   ```
   create AUTHORS
   ```

```
create CHANGELOG
create LICENSE-MIT
create Gruntfile.js
create package.json
create .editorconfig
create README.md
create .gitignore
create .gitattributes
create bower.json
create src/data/site.yml
create src/assets/theme.css
create src/content/markdown.md
create src/templates/pages/blog.hbs
create src/templates/pages/index.hbs
create src/templates/layouts/default.hbs
create src/templates/partials/navbar-fixed-top.hbs
```

7. After creating the aforementioned files the generator should also automatically install the required dependencies mentioned in the newly created `package.json` and `bower.json` files. If however it appears that the dependencies have not been installed, you can try to do so manually using the following command:

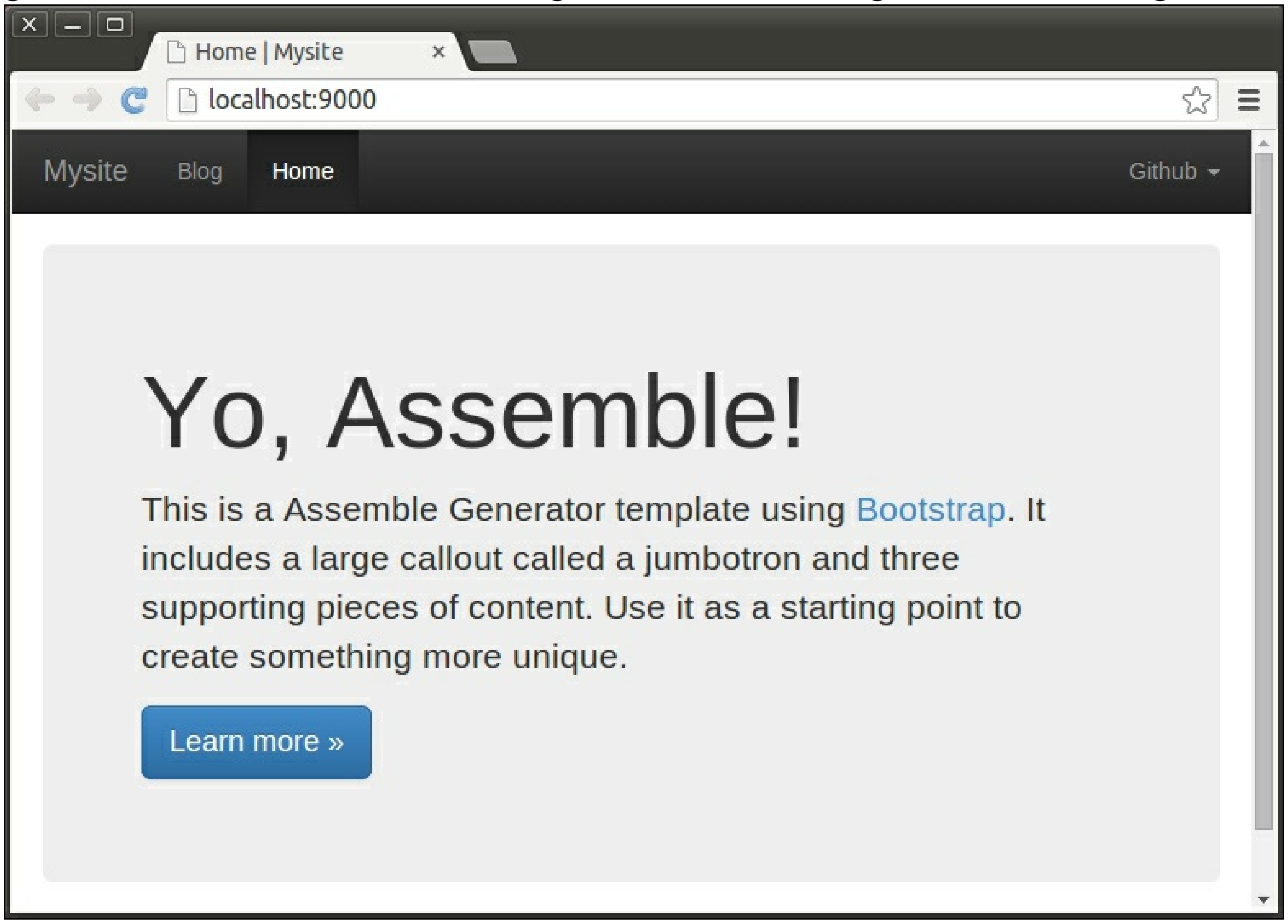
   ```
   $ bower install & npm install
   ```

8. With the necessary files created and dependencies installed, we can now build our site and have it locally hosted, using the `grunt server` command. This should produce output similar to the following:

   ```
   Running "clean:0" (clean) task
   >> 0 paths cleaned.

   Running "copy:bootstrap" (copy) task
   Created 4 directories, copied 12 files

   Running "copy:theme" (copy) task
   Created 1 directories, copied 1 files

   Running "assemble:pages" (assemble) task
   Assembling dist/blog.html OK
   Assembling dist/index.html OK
   >> 2 pages assembled.

   Running "connect:livereload" (connect) task
   Started connect web server on http://localhost:9000

   Running "watch" task
   ```

```
Waiting...
```

9. Once the `server` task has completed, it should automatically open your default browser on `http://localhost:9000/`, which should show you the standard generated site. At the time of writing, it looked something like the following:



Yo, Assemble!

This is a Assemble Generator template using Bootstrap. It includes a large callout called a jumbotron and three supporting pieces of content. Use it as a starting point to create something more unique.

Adding a page to the site

As its core, a static website is just a collection of pages. With our basic setup in place we can look at adding a page that contains some biographical information about us.

Getting started

In this recipe we'll work with the basic project structure that we created in the *Setting up a basic site project* recipe earlier in this chapter. Be sure to refer to it if you are not yet familiar with it's content. Before we start we should also make sure that the site builder and server are running. If they aren't yet running, they can be started using the `grunt server` command.

How to do it...

The following steps take us through adding a page to our static site that will contain our simple biography:

1. First we'll create a file called `bio.hbs` in the `src/templates/pages` directory and fill it with some information about our new page:

   ```
   ---
   title: Bio
   heading: 'Biography'
   ---
   ```

   ### Tip

   The initial part of the file is called the **YAML front matter** section which can be used to provide metadata about a page to the site building tool. You can read more about it at the following URL:

   http://assemble.io/docs/YAML-front-matter.html

2. With our page information added to the file, we can proceed to add the HTML content of the page. Let's do that by adding the following after the front matter section:

   ```
   <div class="jumbotron">
     <h1>{{ heading }}</h1>
     <p>Building awesome sites since 2014.</p>
   </div>
   ```

   The default template language provided by the Assemble project generator is Handlebars, which was used in the preceding code. You can read more about Handlebars at the following URL:

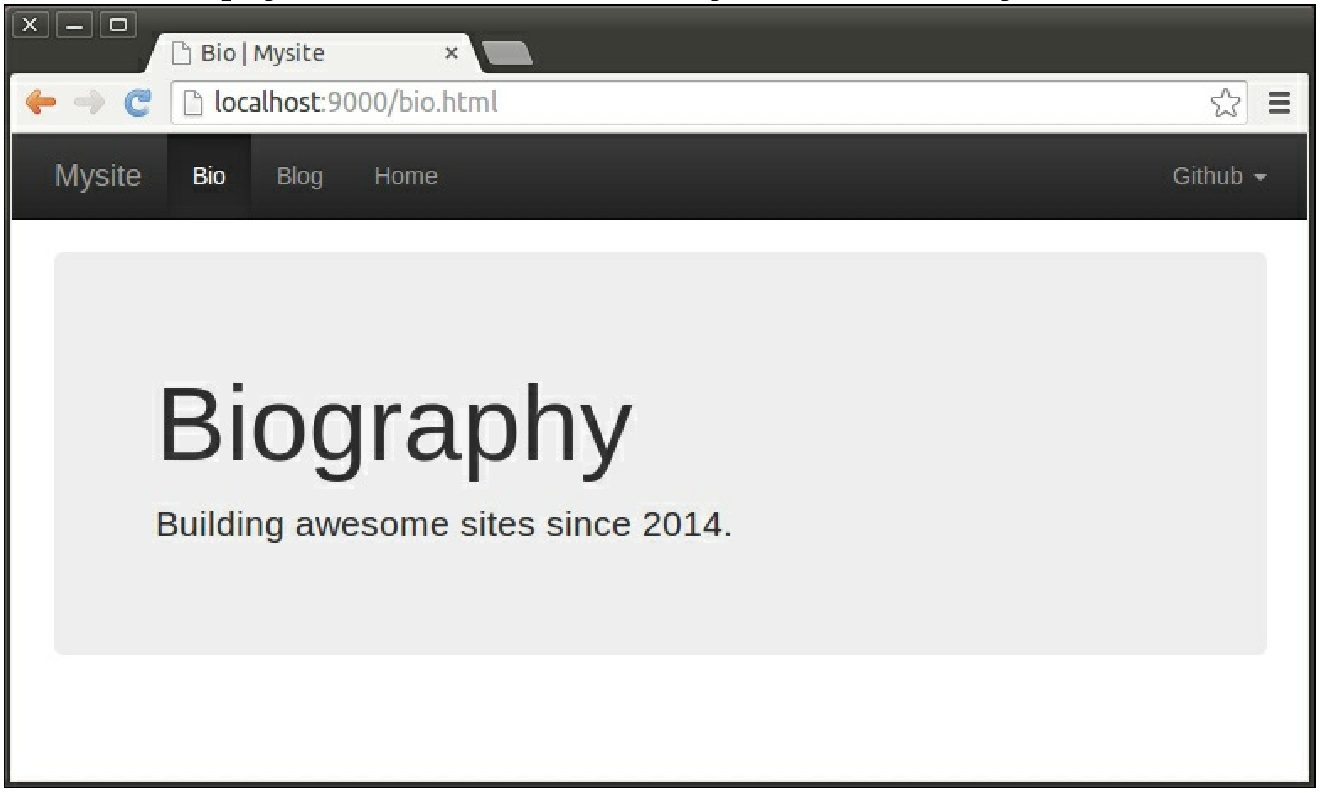
   http://handlebarsjs.com/

   Also be sure to observe that the metadata provided in the front matter section can be used in the page templates. An example of this can be seen with the usage of the `heading` variable in the preceding code.
3. After saving our newly created page file, the site builder that we initially started up should automatically detect that it has been added and proceed to recompile our site, producing output similar to the following:

   ```
   >> File "src/templates/pages/bio.hbs" added
   ```

```
>> File "src/templates/pages/bio.hbs" added.
Running "assemble:pages" (assemble) task
Assembling dist/bio.html OK
Assembling dist/blog.html OK
Assembling dist/index.html OK
>> 3 pages assembled.
```

4. Once it has finished with the compilation it should refresh our browser view of `http://localhost:9000/`, after which we should now see a new `Bio` item in our navigation bar at the top of the site. If we select this item we should be presented with our new page that should look something like the following:



Biography

Building awesome sites since 2014.

Adding content to the site

When developing a website it is usually recommended that the structure and content of a site remain separated. Having it organized in this fashion makes it easy for us to edit the ever changing content of a site, without having to get involved with it's relatively static structure every time we do so.

The **Assemble** framework comes bundled with a library that implements the **Markdown** text formatting syntax; this allows us to easily create content documents with a minimal structure that is rendered to well-formed HTML.

You can read more about the Markdown syntax at following the URL:

http://daringfireball.net/projects/markdown/

Getting started

In this recipe we'll continue working on the project from the *Adding a page to the site* recipe in this chapter. Be sure to refer to it if you are not yet familiar with its contents. Before we start we should also make sure that the site builder and server are running. If they aren't yet running they can be started using the `grunt server` command.

How to do it...

The following steps take us through creating a markdown document to contain our biography and using it in a page.

1. First, we'll create a new file called `bio.md` in the `src/content` directory and fill it with the contents of our biography in Markdown format:

   ```
   ## Achievements

   * Built this awesome site
   * Mastered Grunt
   * Explored Assemble

   ## History

   Born in a galaxy far, far away.
   ```

   ### Tip

   A handy cheatsheet for the Markdown syntax supported by Assemble can be found at the following URL:

   http://assemble.io/docs/Cheatsheet-Markdown.html

2. With our content file created we now need to use it on our site. We'll do that by changing the contents of the `src/templates/pages/bio.hbs` file to the following:

   ```
   ---
   title: Bio
   heading: 'Biography'
   ---
   <div class="jumbotron">
     <h1>{{ heading }}</h1>
     {{md 'src/content/bio.md'}}
   </div>
   ```

   ### Tip

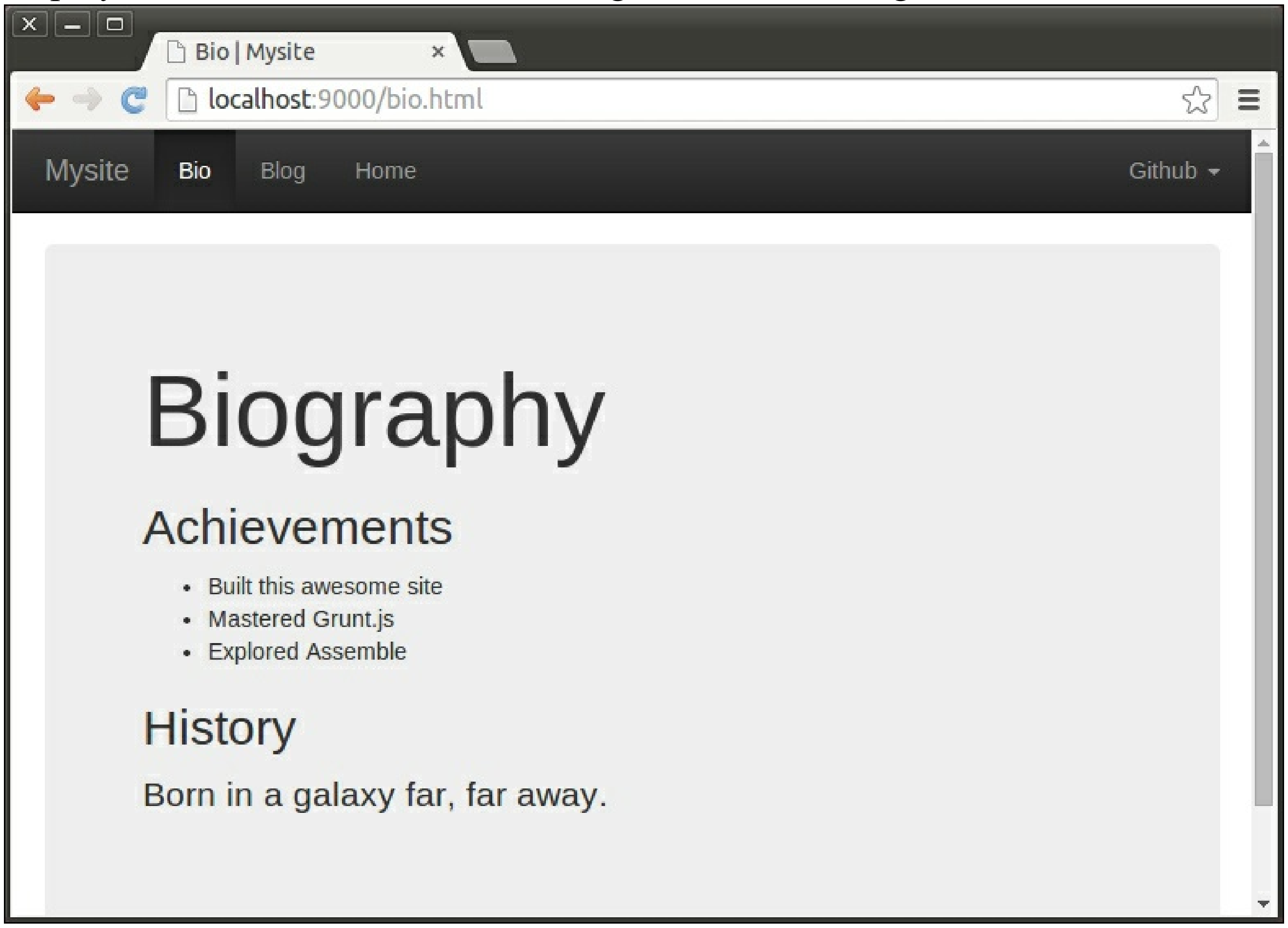
   The `md` **Handlebars** helper is provided by the `handlebars-helpers` package that comes bundled with Assemble. It can be used to render a Markdown file by providing either a string or a variable that contains a file path.

3. After saving the altered `bio.hbs` file, the site builder that we initially started up

should automatically detect that it has been modified and proceed to recompile our site, producing output similar to the following:

```
>> File "src/templates/pages/bio.hbs" changed.
Running "assemble:pages" (assemble) task
Assembling dist/bio.html OK
Assembling dist/blog.html OK
Assembling dist/index.html OK
>> 3 pages assembled.
```

4. If we now have a look at our biography page at `http://localhost:9000/bio.html` we'll see that our new content is now displayed on it; it should look something like the following:



Biography

Achievements

- Built this awesome site
- Mastered Grunt.js
- Explored Assemble

History

Born in a galaxy far, far away.

Adding data to the site

Quite often we'll find that the majority of the content on a site repeats a pattern, and that only certain parts of the content differ from page to page. A site that displays articles for instance, will display each article using the same layout, and also have them listed in a repetitive manner. As developers, we'd probably like to avoid having to create each of these similar structured pages or items by hand.

For this purpose it's usually recommended to store the content we like to display in this fashion as data, which we can then render using a template or layout that will present each of them with the same template.

Getting started

In this recipe we'll continue working with the project from the *Adding content to the site* recipe found earlier in this chapter. Be sure to refer to it if you are not yet familiar with its contents. Before we start we should also make sure that the site builder and server are running. If they aren't yet running they can be started using the `grunt server` command.

How to do it...

The following steps take us through adding data that represents the items in our navigation bar and using that data to customize the rendering of the navigation bar.

1. Due to there only being one navigation bar for our site, we might as well just add it to the existing `site.yml` file located in the `src/data` directory. This file is used for storing data related to the site itself. Let's add the following `sections` collection to its contents:

   ```
   title: Mysite
   sections:
     - title: Home
       dest: 'dist/index.html'
     - title: Bio
       dest: 'dist/bio.html'
     - title: Blog
       dest: 'dist/blog.html'
   ```

   ### Tip

   The preceding file has its data stored in the YAML format. The following URL provides more information about the YAML format at http://yaml.org/.

2. Next we'll be altering the template of the navigation bar to use the `sections` data we've provided in the `site.yml` file. To do this we will first need to find the following block of code in the `navbar-fixed-top.hbs` file located in the `src/templates/partials` directory:

   ```
   {{#each pages}}
   <li{{#if this.isCurrentPage}} class="active"{{/if}}>
     <a href="{{relative dest this.dest}}">{{data.title}}</a>
   </li>
   {{/each}}
   ```

3. Once we've found the aforementioned block of code, we can replace the it with the code that will list navigation items according to the `sections` data. This is done by replacing it with the following code:

   ```
   {{#each site.sections}}
   <li {{#is dest ../page.dest}} class="active"{{/is}}>
     <a href="{{relative ../page.dest dest}}">{{title}}</a>
   </li>
   {{/each}}
   ```

Tip

Here, the `#is` helper is used to check that the full path name of the page is the same as the one that should be linked to by the navigation bar item. If they match, then the item should be rendered as the currently activated one.

The `relative` helper is used here to determine the relative path from the current page to the one that the navigation item should be linked to.

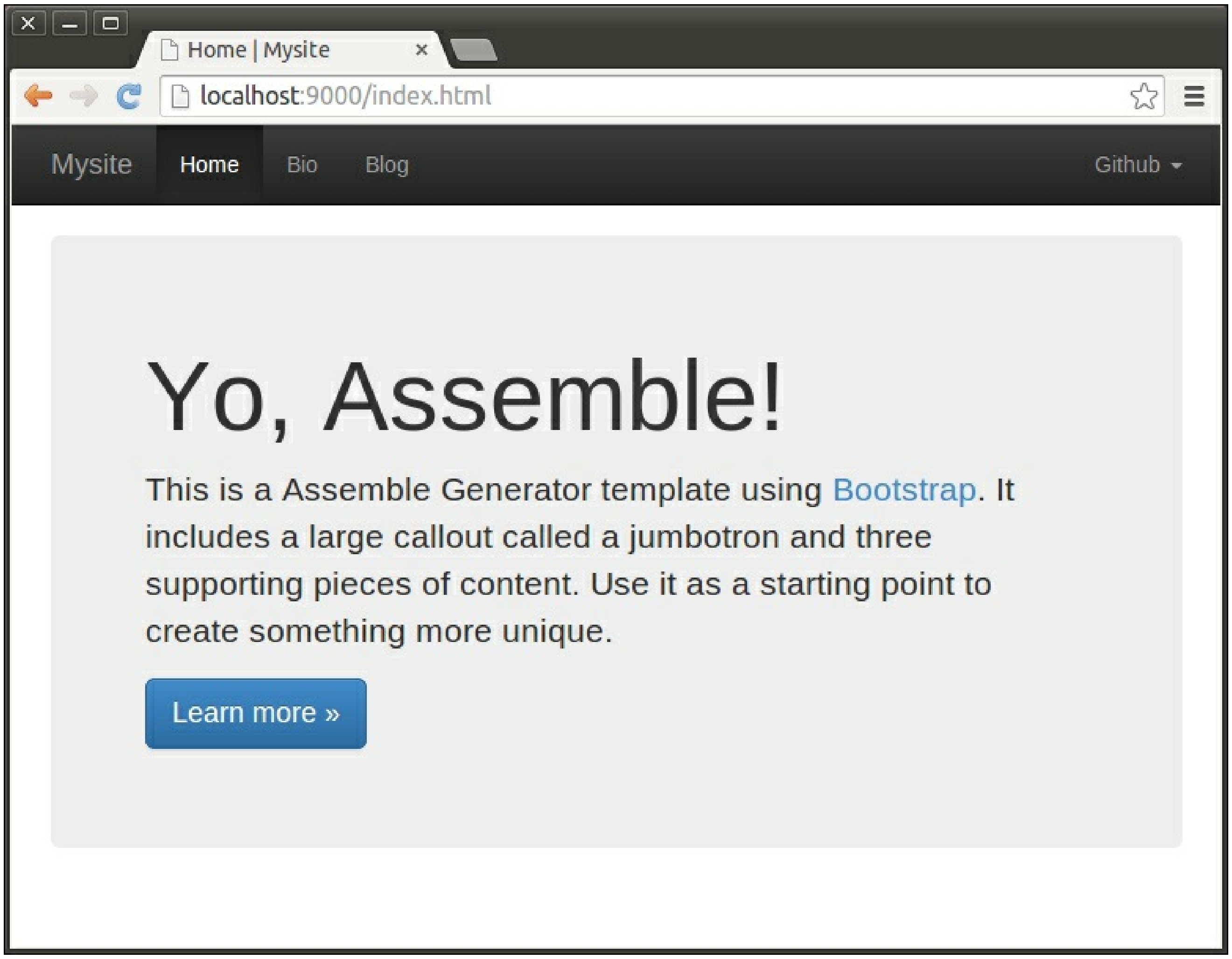
Both the helpers mentioned here are provided by the `handlebars-helpers` library bundled along with Assemble.

The Assemble framework makes the data contained in the `src/data` directory available according to variables with the same names as the filenames the directory contains. This is how we can use the `site` variable name in the previous example to access the data contained in the `site.yml` file

4. After saving the altered `navbar-fixed-top.hbs` file the site builder that we initially started up should automatically detect that it has been modified, and proceed to recompile our site, producing output similar to the following:

   ```
   >> File "src/templates/partials/navbar-fixed-top.hbs" changed.
   Running "assemble:pages" (assemble) task
   Assembling dist/bio.html OK
   Assembling dist/blog.html OK
   Assembling dist/index.html OK
   >> 3 pages assembled.
   ```

5. If we now have a look at any of the pages at `http://localhost:9000/` we'll see that the items in the navigation bar are now ordered according to the order of the items in the `sections` data:



Yo, Assemble!

This is a Assemble Generator template using Bootstrap. It includes a large callout called a jumbotron and three supporting pieces of content. Use it as a starting point to create something more unique.

Creating and using a site layout

Having a consistent layout across a site can drastically improve its usability and also save time in its development. The Assemble framework allows us to easily create and use layouts in various ways.

Getting ready

In this recipe we'll continue working on the project from the *Adding data to the site* recipe in this chapter. Be sure to refer to it if you are not yet familiar with it's contents. Before we start we should also make sure that the site builder and server are running. If they aren't yet running they can be started using the `grunt server` command

How to do it...

The following steps take us through creating a new layout template and setting up some sample blog pages that use it.

1. First, we'll create a new layout template that will provide the layout for our sample blog post pages. Let's create a file called `post.hbs` in the `src/templates/layouts` directory and fill it with the following contents:

```
---
layout: src/templates/layouts/default.hbs
---
<div class="row">
  <div class="col-xs-3">
    <ul class="nav nav-pills nav-stacked">
      {{#each pages}}
        <li{{#if this.isCurrentPage}} class="active"{{/if}}>
          <a href="{{relative dest this.dest}}">
            {{data.title}}
          </a>
        </li>
      {{/each}}
    </ul>
  </div>
  <div class="col-xs-9">
    <h1>{{title}}</h1>
    {{> body}}
  </div>
</div>
```

Tip

At the top of the file in the YAML front matter section, we made use of the special `layout` property to indicate the parent template that should be used for this template. You can read more about how template inheritance works in Assemble at the following web address:

http://assemble.io/docs/Layouts.html

The combination of HTML and CSS used to build this layout is provided by the **Bootstrap** project that is included by default when using the Assemble Yeoman generator we used to set up this project. You can read more about Bootstrap at the following web address:

http://getbootstrap.com/

2. Now we'll create some sample blog pages that will be making use of our newly created layout. We'll do this by creating three files called in the `src/templates/pages/posts` directory with the following names: `one.hbs`, `two.hbs`, and `three.hbs`. Each of these should contain something similar to the following:

```
---
title: One
---
<p>This is post one.</p>
```

Tip

The content of these pages can be pretty much anything, as long as there is at least a front matter section containing a `title` and some HTML content after that.

3. In order for our newly created pages to be rendered we will have to add a new `assemble` task target. This target will however be sharing many of the options of the existing `pages` target, so what we'll do next is move the `options` of the `pages` target into the `assemble` task configuration itself, so that it can be shared across all the targets in the `assemble` task:

```
assemble: {
  options: {
    flatten: true,
    assets: '<%= config.dist %>/assets',
    layout: '<%= config.src %>/templates/layouts/default.hbs',
    data: '<%= config.src %>/data/*.{json,yml}',
    partials: '<%= config.src %>/templates/partials/*.hbs'
  },
  pages: {
    files: {
      '<%= config.dist %>/': [
        '<%= config.src %>/templates/pages/*.hbs'
      ]
    }
  }
}
```

4. With the options shared across all targets now, we can add the `task` target that will be rendering our sample post pages for us. Let's add the following task target called `posts` to the assemble task:

```
posts: {
    options: {
      layout: '<%= config.src %>/templates/layouts/post.hbs',
    },
    files: {
      '<%= config.dist %>/posts/': [
        '<%= config.src %>/templates/pages/posts/*.hbs'
      ]
    }
  }
```

5. In order for us to reach these newly generated pages to make some changes to the blog post listing page. This can be done by changing the contents of the `blog.hbs` file in the `src/templates/pages` directory to the following:

```
---
title: Blog
---
<h1>Posts</h1>
<ul>
  <li><a href="/posts/one.html">One</a></li>
  <li><a href="/posts/two.html">Two</a></li>
  <li><a href="/posts/three.html">Three</a></li>
</ul>
```

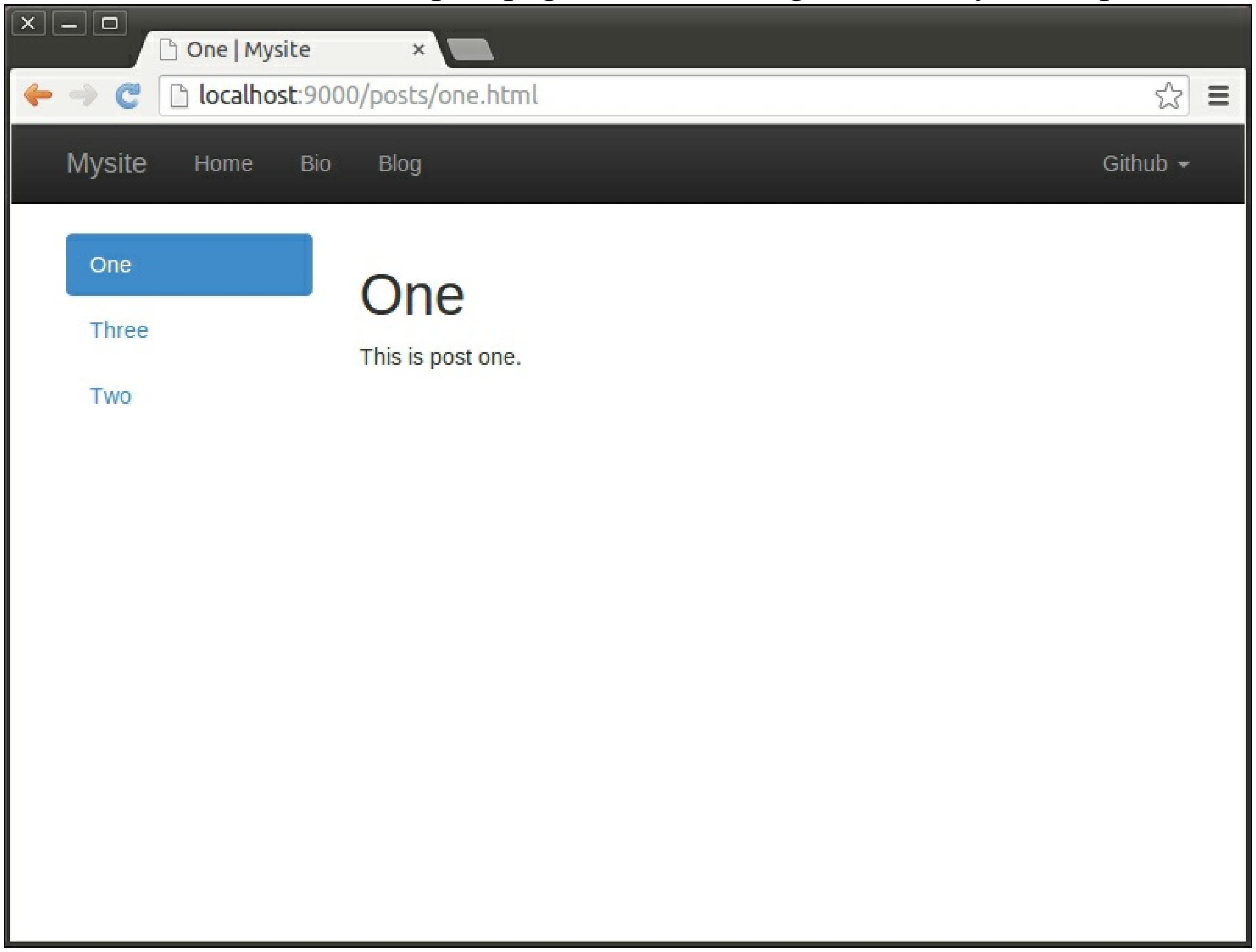
6. Before we continue we'll also have to take care of a slight side effect we caused when placing our sample page templates in the `src/templates/pages/posts` directory. The default `watch` task setup that triggers automatic compilation of changed templates isn't configured to look for changes deeper than the `src/templates/pages` directory itself. This means that changes to our sample page templates won't trigger a re-compilation. Let's take care of this problem by changing the `files` configuration of the `assemble` target of the `watch` task to the following:

```
assemble: {
  files: [
    '<%= config.src %>' +
    '/{content,data,templates}/**/*.{md,hbs,yml}'
  ],
  tasks: ['assemble']
}
```

7. Finally, we can restart our server that was running so that the configuration changes may take effect. After restarting, we should see some extra output with regards to the rendering of our sample post pages:

```
Running "assemble:posts" (assemble) task
Assembling dist/posts/one.html OK
Assembling dist/posts/three.html OK
Assembling dist/posts/two.html OK
>> 3 pages assembled.
```

8. If we now navigate to the blog section of our site at `http://localhost:9000/blog.html` and follow one of the listed links we should now see the selected post page rendered using our new layout template:



One

This is post one.

Generating pages from a collection

In case we've got a collection of items in our site data, we might very well want to generate some pages using that collection. A site that displays blog posts, for instance, would probably want an individual page generated for each post in the collection of posts.

Getting started

In this recipe we'll be continuing work on the project from the *Creating and using a site layout* recipe in this chapter. Be sure to refer to it if you are not yet familiar with it's content. Before we start we should also make sure that the site builder and server are running. If they aren't yet running they can be started using the `grunt server` command.

How to do it...

The following steps take us through creating a data file that contains a collection of our blog posts, and configuring our project to generate pages using that collection.

1. Let's start with creating the collection data that we'll be generating pages from. We can do this by creating a file called `posts.yml` in the `src/data` directory and giving it the following content:

   ```
   - filename: one
     data:
       title: One
     content: <p>This is post one.</p>
   - filename: two
     data:
       title: Two
     content: <p>This is post two.</p>
   - filename: three
     data:
       title: Three
     content: <p>This is post three.</p>
   ```

   ### Tip

   In each of the page objects, the `filename` property indicates the filename of the page that is to be generated, the `data` property can be used to send any kind of metadata to the page, and the `content` property can be used to provide the content that should be rendered by the `body` tag.

   The aforementioned format is determined by the `pages` option for the `assemble` task that will be used later in this recipe.

2. Now that we've got the out blog posts in a data collection, we can remove the page templates we previously used to represent them. This can be done by simply deleting the entire `src/templates/pages/posts` directory.
3. With our collection ready and the old page templates removed, we can now alter our configuration to render the post pages from the collection. We'll do this by making the following changes to the `posts` target in the `assemble` task:

   ```
   Posts: {
     options: {
       layout: '<%= config.src %>/templates/layouts/post.hbs',
       pages: grunt.file.readYAML('src/data/posts.yml')
   ```

```
  },
  files: {'<%= config.dist %>/posts/': []}
}
```

Tip

The collection that should be used to generate the new pages from is indicated using the pages option. As can be seen in the preceding example, we're also making use of the `grunt.file.readYAML` utility to load the collection from the `src/data/posts.yml` file we created earlier.

The destination in the files object no longer indicates specific `files` that should be rendered, but now indicates the directory into which newly generated pages should be saved to.

4. Finally, we can restart our server that was running in the background for the configuration changes may take effect. After restarting, we should see some extra output with regards to the rendering of our post pages from the provided collection:

```
Running "assemble:posts" (assemble) task
Assembling dist/posts/one.html OK
Assembling dist/posts/two.html OK
Assembling dist/posts/three.html OK
>> 3 pages assembled.
```

5. If we now navigate around the blog post items in our site, we should see that they now reflect the data that is being provided by the posts collection.

Creating a template helper

In the process of building page templates, we often come across patterns that are repeated in their rendering logic. Template helpers allow us to wrap one of these identified patterns in such a way that it can be easily reused in our templates across our site.

Arguments can be passed to template helpers, and the logic of the helpers themselves are implemented using a regular programming language, making the possibilities of what can be achieved with them quite numerous.

Getting started

In this recipe we'll be continuing work on the project from the *Generating pages from a collection* recipe. Be sure to refer to it if you are not yet familiar with it's content. Before we start we should also make sure that the site builder and server are running. If they aren't yet running they can be started using the `grunt server` command.

How to do it...

The following steps take us through creating a template tag for automatically listing our posts on our blog page using the data in the posts collection.

1. Let's start by creating the `lib/helpers` directory, which is used to contain the source files that contain the code for our helpers.
2. With our directory ready now, we can create a file called `pages.js` in the `lib/helpers` directory and fill it with the following code:

```
module.exports.register = function(
  Handlebars, options, params
) {
  Handlebars.registerHelper(
    'pages',
    function(target, options) {
      var config = params.grunt.config;
      var pages = config.get('assemble')[target].options.pages;
      var result = '';
      for(var i = 0; i < pages.length; i++) {
        result = result + options.fn(pages[i]);
      }
      return result;
    }
  );
};
```

Tip

The preceding helper uses the name of the task target, provided by the `target` argument, to find the appropriate target configuration. It then extracts the `pages` collection from it and loops over this collection to render the helper's body, with each item as the context.

You can read more about the structure of this file and registering custom helpers at the following URL:

http://assemble.io/docs/Custom-Helpers.html

3. To keep with the convention of our Grunt configuration file, we'll add a property to the `config` task that indicates the name of the directory that will contain reusable libraries. After adding the property, the task should look similar to the following:

```
config: {
  lib: 'lib',
  src: 'src',
  dist: 'dist'
}
```

4. Since we've added more code to our project in the form of the template helpers, we should probably alter our `watch` task so that changes to the template helper code would also trigger a rebuild of our site. This can be done by making the following changes to the `assemble` target in the `watch` task:

   ```
   assemble: {
     files: [
       '<%= config.src %>' +
         '/{content,data,templates}/**/*.{md,hbs,yml}',
       '<%= config.lib %>/helpers/*.js'
     ],
     tasks: ['assemble']
   }
   ```

5. Next we'll alter our configuration to load the helpers in our `lib/helpers` directory, making them available for use in all our templates. This can be done by adding the following `helpers` option to the collection of options for the `assemble` task:

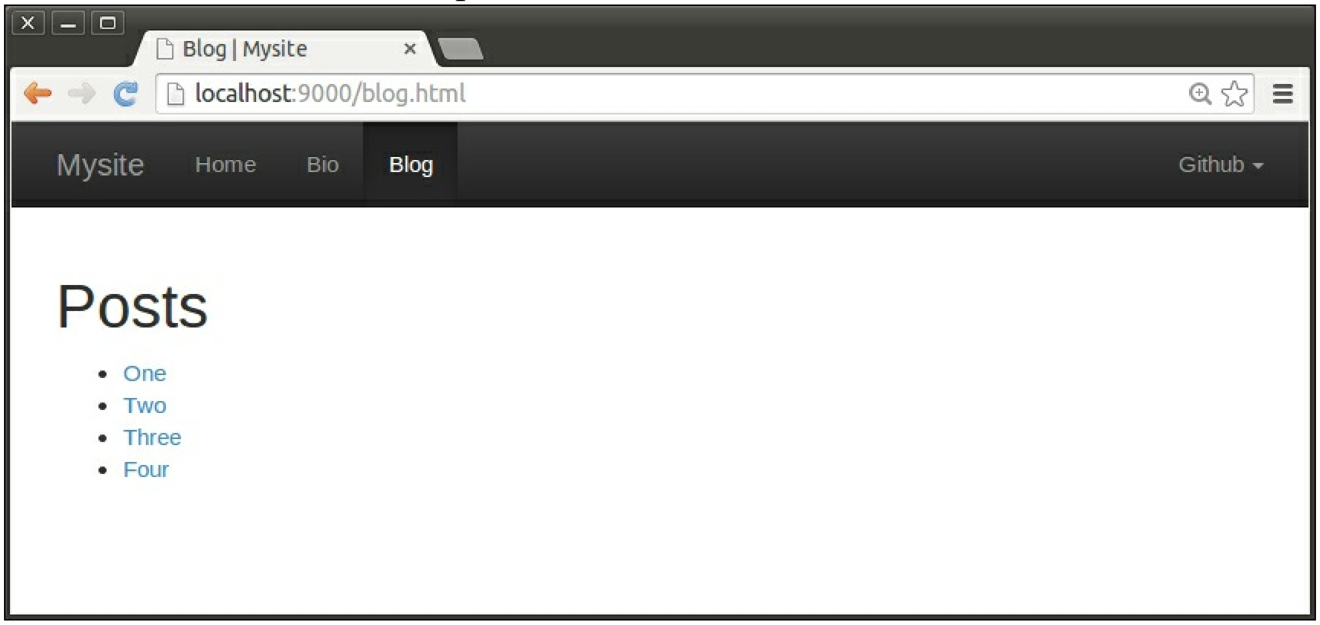
   ```
   options: {
     flatten: true,
     assets: '<%= config.dist %>/assets',
     layout: '<%= config.src %>/templates/layouts/default.hbs',
     data: '<%= config.src %>/data/*.{json,yml}',
     partials: '<%= config.src %>/templates/partials/*.hbs',
     helpers: '<%= config.lib %>/helpers/*.js'
   }
   ```

6. With our template helpers loaded and ready to be used, we can make use of it on our site's blog page to list links to all the posts in our posts collection. We do this by changing the contents of `src/templates/pages/blog.hbs` to the following:

   ```
   ---
   title: Blog
   ---
   <h1>Posts</h1>
   <ul>
     {{#pages 'posts'}}
       <li>
        <a href="/posts/{{filename}}.html">
         {{data.title}}
        </a>
       </li>
   ```

```
  {{/pages}}
</ul>
```

7. Finally we can restart `server` that was running in the background so that the configuration changes can take effect. After restarting we should see that the list rendered on the blog page of the site at `http://localhost:9000/blog.html` now reflects the items in the posts collection:



Posts

- One
- Two
- Three
- Four

Using a plugin

For the purpose of extending its core functionality, the Assemble framework provides a plugin system. There are currently only a handful of plugins available for Assemble, but having the plugin system available makes it easy for us to create and reuse extensions of our own, and perhaps even share them with the community.

Getting ready

In this recipe we'll be continuing working on the project from the *Creating a template helper* recipe. Be sure to refer to it if you are not yet familiar with it's content. Before we start we should also make sure that the site builder and server are running. If they aren't yet running they can be started using the `grunt server` command.

How to do it...

The following steps take us through making use of the `assemble-middleware-sitemap` plugin to generate a well-formed **sitemap** that can be used by web crawlers to navigate and index the site.

1. First, we need to install the package that contains the `assemble-middleware-sitemap` plugin, and save it to our package dependencies. The following command does this for us:

   ```
   $ npm install assemble-middleware-sitemap
   ```

2. Next we'll need to indicate that the plugin should be used by all the targets of the `assemble` tasks. This can be done by adding `plugins` configuration options to the `assemble` task in the Grunt configuration, indicating the name of the newly installed plugin package:

   ```
   options: {
     flatten: true,
     assets: '<%= config.dist %>/assets',
     layout: '<%= config.src %>/templates/layouts/default.hbs',
     data: '<%= config.src %>/data/*.{json,yml}',
     partials: '<%= config.src %>/templates/partials/*.hbs',
     helpers: '<%= config.lib %>/helpers/*.js',
     plugins: ['assemble-middleware-sitemap']
   }
   ```

3. Finally, we can restart our server that was running in the background so that the configuration changes may take effect. After restarting we should see that the `robots.txt` and `site.xml` files have been generated in the `dist` directory.

Index