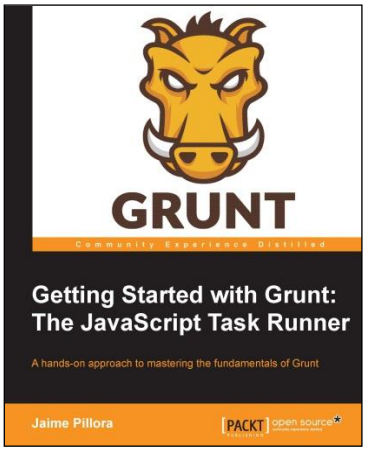
[PACKT] open source
community experience distilled
PUBLISHING

Getting Started with Grunt: The JavaScript Task Runner

Jaime Pillora



Chapter No. 1
"Introducing Grunt"

In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.1 "Introducing Grunt"

A synopsis of the book's content

Information on where to buy this book

About the Author

Jaime Pillora is a passionate full-stack JavaScript developer, an open source advocate and contributor, and is currently the CTO of Luma Networks, a wellfunded networking startup in Sydney, Australia.

Jaime has always been interested in all things computer science, and from a young age, he began to devote his time and effort to learning and perfecting his knowledge in the field. Jaime holds a Bachelor of Computer Science from the University of New South Wales. In all of his work, Jaime strives to reduce technical debt while maintaining maximum efficiency; this is done through software engineering best practices, combined with using the best tools for the given situation. Grunt is one such tool, which is utilized in every frontend project. Jaime's interest in Grunt began early on in his development career and he has since become a renowned expert.

Jaime has been working as a frontend JavaScript developer since 2008, and a backend JavaScript developer utilizing Node.js since 2011. Currently, Jaime leads all software development at Luma Networks, who is implementing software-defined networking on commodity hardware utilizing JavaScript.

I would like to thank my loving partner, Jilarra, for her support during the many hours put into this book, and her contribution to the proofreading and editing of the final drafts.

Getting Started with Grunt: The JavaScript Task Runner

Getting Started with Grunt: The JavaScript Task Runner is an introduction to the popular JavaScript build tool, Grunt. This book aims to provide the reader with a practical skillset, which can be used to solve real-world problems. This book is example driven, so each feature covered in this book is explained and also reinforced through the use of *runnable* examples, this dual method of learning will provide the reader with the means to verify that the theory aligns with its practical use.

All of the software used in this book is open source and when covered, some will be accompanied with a short history while crediting the author. These open source developers do not release their work for monetary gain, instead, they hope to provide utility for others and to forward the community, and for this, they should be duly recognized.

What This Book Covers

Chapter 1, *Introducing Grunt*, explains exactly what Grunt is and why we would want to use it. Then, instead of starting at the very beginning, we temporarily jump ahead to review a set of real-world examples. This gives us a glimpse of what Grunt can do, which will help us to see how we could use Grunt in our current development workflow.

Chapter 2, *Setting Up Grunt*, after finishing our forward escapade, we jump back to the very beginning and start with the two primary technologies surrounding Grunt: Node.js and its package manager—npm. Then, we proceed to installing each of these tools and setting up our first Grunt environment. Next, we learn about the `package.json` and `Gruntfile.js` files and how they are used to configure a Grunt build. We will also cover the various Grunt methods used for configuration and the types of situations where each is useful.

Chapter 3, *Using Grunt*, extends on what we learned in the previous chapter, to the use and creation of tasks that consume our freshly made configuration. We will cover tasks, multitasks, and asynchronous tasks. We look in-depth into the task object and how we can use it effectively to perform common fi le-related actions. Finally, we review running Grunt tasks and methods that customize Grunt execution to our benefit.

Chapter 4, *Grunt in Action*, begins with an empty folder and gradually constructs a Grunt environment for a web application. Throughout this process, we use various examples from *Chapter 1*, *Introducing Grunt*, make use of the configuration strategies from *Chapter 2*, *Setting Up Grunt*, and include some extra features from *Chapter 3*, *Using Grunt*. At the end of this chapter, we shall be left with a Grunt environment that compiles and optimizes our CoffeeScript, Jade, and Stylus, and deploys our resulting web application to Amazon's S3.

Chapter 5, *Advanced Grunt*, introduces some of the more advanced use cases for Grunt; these introductions are intended to be purely an entry to each topic while providing the resources to learn more. We briefly cover testing with Grunt, Grunt plugins, advanced JavaScript, development tools and more.

1
Introducing Grunt

In this chapter, we will first define Grunt and cover some of the reasons why we would want to use it. Then, instead of starting at the beginning, we'll temporarily jump ahead to review some real-world use cases. Each example will contain a brief summary, but it won't be covered in detail, as the purpose is to provide a glimpse of what is to come. These examples will also provide us with a general understanding of what to expect from Grunt and hopefully, with this sneak peak, an idea of how Grunt's power and simplicity could be applied to our own projects.

What is Grunt?

When Ben Alman released Grunt (`http://gswg.io#grunt`) in March 2012, he described it as a task-based command line build tool for JavaScript projects. Now, with the release of Grunt version 0.4.x, the project caption is The JavaScript Task Runner. Build tools or task runners are most commonly used for automating repetitive tasks, though we will see that the benefits of using Grunt far exceed simple automation.

The terms build tool and task runner essentially mean the same thing and throughout this book, I will always use build tool, though both can be used interchangeably. Build tools are programs with the sole purpose of executing code to convert some source code into a final product, whether it be a complete web application, a small JavaScript library or even a Node.js command-line tool. This build process can be composed of any number of steps, including: style and coding practice enforcement, compiling, file watching and automatic task execution, and unit testing and end-to-end testing, just to name a few.

Grunt has received huge success in the open-source community, especially with the rise of JavaScript following the world's increasing demand for web applications. At the time of writing this book (December 2013), Grunt is downloaded approximately 300,000 times per month (`http://gswg.io#grunt-stats`) and the open-source community has published approximately 1,400 Grunt plugins in npm (the Node.js package manager `http://gswg.io#npm`) and these numbers continue to rise.

Node.js (`http://gswg.io#node`) is a platform for writing JavaScript command-line tools, which run on all major operating systems. Grunt is one such command-line tool. Once installed, we can execute `grunt` on the command line. This tells Grunt to look for a `Gruntfile.js` file. This choice of name refers to the build tool **Make**, and its `Makefile`. This file is the entry point to our build, which can define tasks inline, load tasks from external files, load tasks from external modules, and configure these tasks and much more.

Let's briefly review a simple example of a `Gruntfile.js` file so we can get a glimpse of what is to come:

```
//Code example 01-minify
module.exports = function(grunt) {

  // Load the plugin that provides the "uglify" task.
  grunt.loadNpmTasks('grunt-contrib-uglify');

  // Project configuration.
  grunt.initConfig({
    uglify: {
      target1: {
        src: 'foo.js',
        dest: 'foo.min.js'
      }
    }
  });

  // Define the default task
  grunt.registerTask('default', ['uglify']);
};
```

In this short example, we are using the `uglify` plugin to create a minified (or compressed) version of our main project file—`foo.js` in this case. First, we load the plugin with `loadNpmTasks`. Next, we'll configure it by passing a configuration object to `initConfig`. Finally, we'll define a `default` task, which in this example, is simply an alias to the `uglify` task.

Now, we can run the `default` task with `grunt` and we should see the following output:

```
$ grunt
Running "uglify:target1" (uglify) task
File "foo.min.js" created.
Done, without errors.
```

We've just created and successfully run our first Grunt build!

Why use Grunt?

In the past five years, due to the evolution of Web browsers, focus has shifted from Desktop applications to Web applications. More companies are realizing that the web is a perfect platform to create tools to save people time and money by providing quick and easy access to their service. Whether it is ordering pizza or Internet banking, web applications are fast becoming the platform of choice for the modern business. These modern companies know that, if they were to build an application for a specific platform, like the iOS or Windows operating systems, they would be inherently restricting their audience, as each operating system has its own percentage of the total user base. They've realized that in order to reach *everyone*, they need a ubiquitous platform that exists in all operating systems. This platform is the Web. So, if everyone with Internet access has a browser, then by targeting the browser as our platform, our potential user base becomes everyone on the Internet.

The Google product line is a prime example of a business successfully utilizing the browser platform. This product line includes: Google Search, YouTube, Gmail, Google Drive, Google Docs, Google Calendar, and Google Maps. However, providing a rich user experience comes with a cost. These applications are tremendously more complex than a traditional website made with jQuery animated menus.

Complex JavaScript Web Applications require considerable design and planning. It is quite common for the client-side (or browser) JavaScript code to be more complicated than the server-side code. With this in mind, we need to ensure our code base is manageable and maintainable. The key to code manageability and maintainability is to *logically structure our project* and to *keep our code DRY*. Structuring includes the file and directory structure as well as the code structure (that is HTML, CSS, and JavaScript structure). Maintaining a logical directory structure provides predefined locations for all types of files. This allows us to always know where to put our code, which is very important for rapid development. DRY stands for Don't Repeat Yourself (`http://gswg.io#dry`). Hence, to keep your code DRY is to write code where there is little or no repetition and we embrace the idea of a "single source of truth". Similarly, we want to avoid repetition surrounding our build process. As we'll see throughout this book, Grunt is a great tool for achieving these goals.

Benefits of Grunt

Many people are of the opinion that the benefit of using Grunt (or any build tool for that matter) is to possibly save time, and often this tradeoff—of learning time versus actual development time is deemed too risky, which then leads to the programmer staying safe with the manual method. This perception is misguided. The added efficiency is only one of benefits of using Grunt, the other main benefits include: build consistency, increased effectiveness, community utilization, and task flexibility.

Efficiency

Hypothetically, let's say it takes us 2 minutes per build and we need to build (and run the tests) numerous times every hour, resulting in approximately 50 builds per day. With this schedule, it costs us approximately 100 minutes per day in order to perform the monotonous task of manual running various sets of command-line tools in the right sequence. Now, if learning a new build tool like Grunt takes us 2-3 hours of research and 1-2 hours to implement the existing build process as a Grunt build, then this cost will be recovered in only a week of work. Considering that most programmers will be using their trade for years to come, the decision is simple—use a build tool as it is well worth the time investment.

With this in mind, we can see the time spent to learn a new tool like Grunt is negligible in comparison with the time saved across the entire span of all projects in which that tool is used.

Consistency

The human propensity for error is an unavoidable hurdle programmers face when carrying out a manual build process. This propensity is further increased if a given build process involves each command being manually typed out instead of saving them in some kind of script for easier execution. Even with an array of scripts, problems can still arise if someone forgets to execute one, or if the special script required for a special situation is forgotten.

Using Grunt provides us with the ability to implement our build logic inside the build process. Once the build has been set up and confirmed, this effectively removes the possibility for human error from the equation entirely. This ability also helps newcomers contribute to your projects by allowing them to quickly get started on the code base as opposed to getting bogged down trying to understand the build.

Also, as a result of the great effort behind the Node.js project, we can also run our encapsulated build process across all major operating systems. This allows developers from all walks of life to use and enhance a common build process.

Effectiveness

As well as saving time from doing less, we also save time by staying in the zone. For many programmers, it often takes us some time to gather momentum in order to bring our brains into gear. By automating the build process, we multi-task less, allowing us to keep our minds focused on the current task at hand.

Community

A common problem for many build tools is the lack of community support. Most build tool have plugins for many common build processes, but as soon as we want to perform a task that is too niche or too advanced, we are likely to be forced to restart from scratch.

At the time of writing, npm (the Node.js Package Manager) contained approximately 50,000 modules and, as mentioned above, approximately 1,900 of these are Grunt plugins. These plugins cover a wide array of build problems and are available now via the public npm repository, which provides a purposefully simple means to publish new modules to the repository. As a result of this simplicity, anyone may share their Grunt plugin with the rest of the world with a single `npm publish` command. This concept makes it easy for programmers of every skill level to share their work. Allowing everyone to build upon everyone else's work creates a synergistic community, where the more people contribute, the more valuable the community becomes, which in turn provides further incentive for people to contribute. So, by using Grunt, we tap into the power of the Node.js community. This fact alone should be enough to convince us to use Grunt.

GitHub (`http://gswg.io#github`) is another valuable community tool that greatly benefits Grunt. As of June 2013, JavaScript code makes up 21 percent of code on GitHub making it the most popular programming language on GitHub. However, this fact alone is not the only reason to host your project on GitHub. The Git (`http://gswg.io#git`) **Distributed Version Control System** (**DVCS**) provides the ability to branch and merge code, and the flexibility of both local and remote repositories. This makes it the superior choice for open-source collaboration, compared to other (non-distributed) VCS tools such as SVN or TFS.

With the combination of GitHub (being a great JavaScript open-source collaboration platform) and npm (being so widespread and simple to use) the Grunt team provides Grunt users with the perfect environment for an open-source community to thrive.

We'll cover more on npm in the *Chapter 2*, *Setting Up Grunt* and contributing to open-source projects in *Chapter 5*, *Advanced Grunt*.

Flexibility

Another common problem for many build tools is the level of prior knowledge required to write your own task. Often, they also require varying levels of setup before you can start actually writing code. A Grunt task is essentially just a JavaScript function, and that's it. Tasks can be defined with various levels of complexity to suit the needs of build process. However, remaining at the root of all tasks is the idea of one task being one function—for example, this `Gruntfile.js` defines a simple task called `foo`:

```
//Code example 02-simple-task
module.exports = function(grunt) {

  grunt.registerTask('foo', function() {
    grunt.log.writeln('foo is running...');
  });

};
```

Our new `foo` task is runnable with the command: `grunt foo`. When executed, we see:

```
$ grunt foo
Running "foo" task
foo is running...
```

We'll learn more about Grunt tasks in *Chapter 3*, *Using Grunt*.

The arguments for using various build tools generally stem from two conflicting sides: the simplicity of configuration or the power of scripting. With Grunt however, we get the best of both worlds. We are able to easily create arbitrary tasks as well as define verbose configuration. The following `Gruntfile.js` file demonstrates this:

```
//Code example 03-simple-config
module.exports = function(grunt) {

  grunt.initConfig({
    bar: {
      foo: 42
    }
  });

  grunt.registerTask('bar', function() {
    var bar = grunt.config.get('bar');
    var bazz = bar.foo + 7;
```

```
    grunt.log.writeln("Bazz is " + bazz);
  });
};
```

In this example, we are first initializing the configuration with an object. Then, we are registering a simple task, which uses this configuration. Note, instead of using `grunt.initConfig(…)` in the preceding code, we could also use `grunt.config.set('bar', { foo: 42 });` to achieve the same result.

When we run this example with `grunt bar`, we should see:

```
$ grunt bar
Running "bar" task
Bazz is 49
```

This example demonstrates the creation of a simple task using minimal configuration. Imagine we have created a task which parses JavaScript source code into a tree of syntax nodes, traverses these nodes, performing arbitrary transforms on them (like shortening variable names) and writes them back out to a file, with the ultimate effect of compressing our source code. This is exactly what the `UglifyJS` library does, with many configuration options to customize its operation. We'll cover more on *JavaScript Minification* in the next section.

Real-world use cases

Hearing about the benefits of Grunt is all well and good, but what about actual use cases that the average web developer will face every day in the real world? In this section, we'll take an eagle-eye view of the most common use cases for Grunt.

These examples make use of configuration targets. Essentially, targets allow us to define multiple configurations for a task. We'll cover more on configuration targets in *Chapter 2, Setting Up Grunt*.

Static analysis or Linting

In programming, the term **linting** is the process of finding *probable* bugs and/or style errors. Linting is more popular in dynamically typed languages as type errors may only be resolved at runtime. Douglas Crockford popularized JavaScript linting in 2011 with the release of his popular tool, JSLint.

JSLint is a JavaScript library, so it can be run in Node.js or in a browser. JSLint is a set of predetermined rules that enforce correct JavaScript coding practices. Some of these rules may be optionally turned on and off, however, many cannot be changed. A complete list of JSLint rules can be found at `http://gswg.io#jslint-options`.

This leads us to JSHint. Due to Douglas Crockford's coding style being too strict for some, *Anton Kovalyov* has forked the JSLint project to create a similar, yet more lenient version, which he aptly named: JSHint.

I am a fan of *Douglas Crockford* and his book, *JavaScript – The Good Parts* (`http://gswg.io#the-good-parts`), but like Anton, I prefer a more merciful linter, so in this example below, we will use the Grunt plugin for JSHint: `http://gswg.io#grunt-contrib-jshint`.

```
//Code example 04-linting
//Gruntfile.js
module.exports = function(grunt) {

  // Load the plugin that provides the "jshint" task.
  grunt.loadNpmTasks('grunt-contrib-jshint');

  // Project configuration.
  grunt.initConfig({
    jshint: {
      options: {
        curly: true,
        eqeqeq: true
      },
      target1: ['Gruntfile.js', 'src/**/*.js']
    }
  });

  // Define the default task
  grunt.registerTask('default', ['jshint']);

};

//src/foo.js
if(7 == "7") alert(42);
```

In the preceding code, we first load the `jshint` task. We then configure JSHint to run on the `Gruntfile.js` file itself, as well as all of the `.js` files in the `src` directory and its subdirectories (which is `src/foo.js` in this case). We also set two JSHint options: `curly`, which ensures that curly braces are always used in `if`, `for`, and `while` statements; and `eqeqeq`, which ensures that strict equality `===` is always used.

JSHint has retained most of the optional rules from JSLint and it has also added many more. These rules can be found at: `http://gswg.io#jshint-options`.

Finally, we can run the `jshint` task with `grunt`, and we should see the following:

```
$ grunt
Running "jshint:target1" (jshint) task
Linting src/foo.js...ERROR
[L1:C6] W116: Expected '===' and instead saw '=='.
if(7 == "7") alert(42);
Linting src/foo.js...ERROR
[L1:C14] W116: Expected '{' and instead saw 'alert'.
if(7 == "7") alert(42);

Warning: Task "jshint:target1" failed. Use --force to continue.

Aborted due to warnings.
```

The result shows that JSHint found two warnings in the `src/foo.js` file on:

- Line 1, column 6—since we've enforced the use of strict equality, `==` is not allowed, so it must be changed to `===`.
- Line 1, column 14—since we've enforced the use of the curly braces, the `if` statement body must explicitly use curly braces.

Once we've fixed these two issues as follows:

```
if(7 === "7") {
  alert(42);
}
```

We can then re-run `grunt` and we should see:

```
$ grunt
Running "jshint:target1" (jshint) task
>> 2 files lint free.

Done, without errors.
```

Notice that two files were reported to be lint free. The second file was the `Gruntfile.js` file, and if we review this file, we see it does not break either of the two rules we enabled.

In summary, JSHint is very useful as the first step of our Grunt build as it can help catch simple errors, such as unused variables or accidental assignments in `if` statements. Also, by enforcing particular coding standards on the project's code base, it helps maintain code readability, as all code entering the shared repository will be normalized to a predetermined coding style.

Transcompilation

Transcompiling—also known as source-to-source compilation and often abbreviated to transpiling—is the process of converting the source code of one language to the source code of another. Within the web development community in recent years, there has been an increase in the use of transcompile languages such as Haml, Jade, Sass, LESS, Stylus, CoffeeScript, Dart, TypeScript, and more.

The idea of transcompiling has been around since the 1980s. A popular example was an original C++ compiler (Cfront) by Bjarne Stroustrup, which converted C++ (known as C with Classes at the time) to C.

CoffeeScript

CoffeeScript (`http://gswg.io#coffeescript`) is the most popular transpile language for JavaScript. It was released in 2009 by *Jeremy Ashkenas* and is now the 10th most popular language on GitHub with 3 percent of the all code in public Git repositories. Due to this popularity, a particularly common use case for the modern web developer is to compile CoffeeScript to JavaScript. This can be easily achieved with the Grunt plugin `http://gswg.io#grunt-contrib-coffee`.

In the following example, we'll use the `grunt-contrib-coffee` plugin to compile all of our CoffeeScript files:

```
//Code example 05-coffeescript
module.exports = function(grunt) {

  // Load the plugin that provides the "coffee" task.
  grunt.loadNpmTasks('grunt-contrib-coffee');

  // Project configuration.
  grunt.initConfig({
    coffee: {
      target1: {
        expand: true,
        flatten: true,
        cwd: 'src/',
        src: ['*.coffee'],
        dest: 'build/',
        ext: '.js'
      },
      target2: {
        files: {
          'build/bazz.js': 'src/*.coffee'
        }
```

```
        }
      }
    });

    // Define the default task
    grunt.registerTask('default', ['coffee']);
};
```

Inside the configuration, the `coffee` object has two properties; each of which defines a target. For instance, we might wish to have one target to compile the application source and another target to compile the unit test source. We'll cover more on tasks, multitasks, and targets in *Chapter 2*, *Setting Up Grunt*.

In this case, the `target1` target will compile each `.coffee` file in the `src` directory to a corresponding output file in the `build` directory. We can execute this target explicitly with `grunt coffee:target1`, which should produce the result:

```
$ grunt coffee:target1
Running "coffee:target1" (coffee) task
File build/bar.js created.
File build/foo.js created.

Done, without errors.
```

Next, `target2` will compile and combine each of the `.coffee` files in the `src` directory to a *single* file in the `build` directory called `bazz.js`. We can execute this target with `grunt coffee:target2`, which should produce the result:

```
grunt coffee:target2
Running "coffee:target2" (coffee) task
File build/bazz.js created.

Done, without errors.
```

Combining multiple files into one has advantages and disadvantages, which we shall review in the next section *Minification*.

Jade

Jade (`http://gswg.io#jade`) compiles to HTML and, as with CoffeeScript to JavaScript, Jade has the semantics of HTML, though different syntax. *TJ Holowaychuk*, an extremely prolific open-source contributor, released Jade in July 2010.
More information on the Grunt plugin for Jade can be found at `http://gswg.io#grunt-contrib-jade`.

We'll also notice the following example `Gruntfile.js` file is quite similar to the previous CoffeeScript example. As we will see with many Grunt plugins, both these examples define some kind of transform from one set of source files to another set of destination files:

```
//Code example 06-jade
module.exports = function(grunt) {

  // Load the plugin that provides the "jade" task.
  grunt.loadNpmTasks('grunt-contrib-jade');

  // Project configuration.
  grunt.initConfig({
    jade: {
      target1: {
        files: {
          "build/foo.html": "src/foo.jade",
          "build/bar.html": "src/bar.jade"
        }
      }
    }
  });

  // Define the default task
  grunt.registerTask('default', ['jade']);
};
```

In this example, `target1` will do a one-to-one compilation, where `src/foo.jade` and `src/bar.jade` will be compiled into `build/foo.html` and `build/bar.html` respectively. As we have set the `default` task to be the `jade` task, we can run all of jade's targets with a simple `grunt` command, which should produce:

```
$ grunt
Running "jade:target1" (jade) task
File "build/foo.html" created.
File "build/bar.html" created.

Done, without errors.
```

Stylus

Stylus (`http://gswg.io#stylus`) compiles to CSS, and as before, it has the semantics of CSS though different syntax. *TJ Holowaychuk* also created Stylus, which he officially released in February 2011. More information on the Stylus Grunt plugin can be found at `http://gswg.io#grunt-contrib-stylus`. Similarly to the examples above, the following example `Gruntfile.js` file contains only slight differences. Instead of `jade`, we're configuring `stylus`, and instead of transpiling `.jade` to `.html`, we're transpiling `.styl` to `.css`:

```
//Code example 07-stylus
module.exports = function(grunt) {

  // Load the plugin that provides the "stylus" task.
  grunt.loadNpmTasks('grunt-contrib-stylus');

  // Project configuration.
  grunt.initConfig({
    stylus: {
      target1: {
        files: {
          "build/foo.css": "src/foo.styl"
        }
      }
    }
  });

  // Define the default task
  grunt.registerTask('default', ['stylus']);
};
```

When we run `grunt`, we should see the following:

```
$ grunt
Running "stylus:target1" (stylus) task
File build/foo.css created.

Done, without errors.
```

Haml, Sass, and LESS

Grunt plugins that transpile code are very similar, as previously seen with CoffeeScript, Jade and Stylus. In some way or another, they define a set of input files and a set of output files, and also provide options to vary the compilation. For the sake of brevity, I won't go through each one, but instead I'll provide links to each preprocessor (transcompiler tool) and its respective Grunt plugins:

- Haml—`http://gswg.io#haml—gswg.io#grunt-haml`
- Sass—`http://gswg.io#sass—gswg.io#grunt-contrib-sass`
- LESS—`http://gswg.io#less—gswg.io#grunt-contrib-less`

At the end of the day, the purpose of using transcompile languages is to improve our development workflow, not to hinder it. If using these tools requires a lengthy setup for each, then the more tools we add to our belt, the longer it'll take our team to get up and running. With Grunt, we add each plugin to our `package.json` and with one `npm install` command, we have all the plugins we need and can start transpiling in minutes!

Minification

As web applications increase in complexity, they also increase in size. They contain more HTML, more CSS, more images, and more JavaScript. To provide some context, the uncompressed development version of the popular JavaScript library, jQuery (v1.9.1), has reached a whopping 292 KB. With the shift to mobile, our users are often on unreliable connections and loading this uncompressed jQuery file could easily take more than 5 seconds. This is only one file, however, often websites can be as large as 2-3MB causing load times to skyrocket. A blog post from KISSmetrics (`http://gswg.io#loading-time-study`) reveals the following, using data from `gomez.com` and `akamai.com`:

> *73% of mobile Internet users say they have encountered a website that was too slow to load.*
>
> *51% of mobile Internet users say they have encountered a website that crashed, froze, or received an error.*
>
> *38% of mobile Internet users say they have encountered a website that wasn't available.*
>
> *47% of consumers expect a web page to load in 2 seconds or less.*
>
> *40% of people abandon a website that takes more than 3 seconds to load.*

A 1 second delay in page response can result in a 7% reduction in conversions.

If an e-commerce site is making $100,000 per day, a 1 second page delay could potentially cost you $2.5 million in lost sales every year.

Based on this information, it is clear we should do all we can to reduce page load times. However, manually minifying all of our assets is time consuming, so it is Grunt to the rescue! The Grunt team has plugins for the following common tasks:

- Minify JavaScript—`http://gswg.io#grunt-contrib-uglify`
- Minify CSS—`http://gswg.io#grunt-contrib-cssmin`
- Minify HTML—`http://gswg.io#grunt-contrib-htmlmin`

In the following example `Gruntfile.js`, we see how easy this process is. Much like the compilation tasks above, these minification tasks are also a transformation, in that they have file inputs and file outputs. In this example, we'll utilize the `grunt-contrib-uglify` plugin, which will provide the `uglify` task:

```
grunt.initConfig({
  uglify: {
    target1: {
      src: 'foo.js',
      dest: 'foo.min.js'
    }
  }
});
```

This is only a portion of `Code example 01-minify`, the complete snippet can be found in the code examples (`http://gswg.io#examples`) or by returning to the start of this chapter. As with the `uglify` task, the `cssmin` and `htmlmin` tasks also have options to customize the way our code is compressed. See the corresponding GitHub project pages for more information.

If you're using Jade to construct your HTML, then you can use its built-in compression option by setting `pretty` to `false`.

Concatenation

As with minification, concatenation (or joining) also helps reduce page load time. As per the HTTP 1.1 specification, browsers can only request two files at once (see HTTP 1.1 Pipelining). Although newer browsers have broken this rule and will attempt to load up to six files at once, we will see it is still the cause of slower page load times.

For example, if we open Chrome Developer Tools inside Google Chrome, view the Network tab, then visit the `cnn.com` website, we see approximately 120 file requests, 40 of which are loading from the `cnn.com` domain. Hence, even with six files being loaded at once, our browsers still must wait until a slot opens up before they can start downloading the next set of files.

Also, if there are more files to load over a longer period of time, there will be a higher chance of TCP connection dropouts, resulting in even longer waits. This is due to the browser being forced to re-establish a connection with the server.

When building a large Web Application, JavaScript will be used heavily. Often, without the use of concatenation, developers decide not to segregate their code into discrete modular files, as they would then be required to enter a corresponding script tag in the HTML. If we know all of our files will be joined at build-time, we will be more liberal with creation of new files, which in turn will guide us toward a more logical separation of application scope.

Therefore, by concatenating assets of similar type together, we can reduce our asset count, thereby increasing our browser's asset loading capability.

Although concatenation was solved decades ago with the Unix command: `cat`, we won't use `cat` in this example, instead, we'll use the Grunt plugin: `http://gswg.io#grunt-contrib-concat`. This example `Gruntfile.js` file demonstrates use of the `concat` task, which we'll see is very similar to the tasks above as it is also a fairly simple transformation:

```
//Code example 08-concatenate
module.exports = function(grunt) {

  // Load the plugin that provides the "concat" task.
  grunt.loadNpmTasks('grunt-contrib-concat');

  // Project configuration.
  grunt.initConfig({
    concat: {
      target1: {
        files: {
          "build/abc.js": ["src/a.js", "src/b.js", "src/c.js"]
        }
      }
    }
  });

  // Define the default task
```

```
  grunt.registerTask('default', ['concat']);
};
```

As usual, we will run it with `grunt` and should see the following:

```
$ grunt
Running "concat:target1" (concat) task
File "build/abc.js" created.

Done, without errors.
```

Just like that, our three source files have been combined into one, in the order we specified.

Deployment

Deployment is one of the lengthier tasks when it comes to releasing the final product. Generally, it involves logging into a remote server, manually finding the correct files to copy, restarting the server and praying we didn't forget anything. There may also be other steps involved which could further complicate this process, such as performing a backup of the current version or modifying a remote configuration file. Each one of these steps can be catered for with Grunt, either with plugins, which provide useful tasks, or with our own custom tasks where we may wield the complete power of Node.js.

As mentioned in the first section, we can use Grunt to script these types of processes, thus removing the element of human error. Human error is probably the most dangerous at the deployment step because it can easily result in server down time, which will often result in monetary losses.

In the following subsections, we'll cover three common methods of deploying files to our production servers: FTP, SFTP, and S3. We won't however, cover the creation of custom tasks and plugins in this section, as we will go through these topics in depth in *Chapter 3*, *Using Grunt*.

FTP

The File Transfer Protocol specification was released in 1980. Because of FTP's maturity and supremacy, FTP became the standard way to transfer files across the Internet. Since FTP operates over a TCP connection, and given the fact that Node.js excels in building fast network applications, an FTP client has been implemented in JavaScript in approximately 1000 lines, which is tiny! It can be found at `http://gswg.io#jsftp`.

A Grunt plugin has been made using this implementation, and this plugin can be found at `http://gswg.io#grunt-ftp-deploy`. In the following example, we'll use this plugin along with a local FTP server:

```
//Code example 09-ftp
module.exports = function(grunt) {

  // Load the plugin that provides the "ftp-deploy" task.
  grunt.loadNpmTasks('grunt-ftp-deploy');

  // Project configuration.
  grunt.initConfig({
    'ftp-deploy': {
      target1: {
        auth: {
          host: 'localhost',
          port: 21,
          authKey: 'my-key'
        },
        src: 'build',
        dest: 'build'
      }
    }
  });

  // Define the default task
  grunt.registerTask('default', ['ftp-deploy']);
};
```

When the `ftp-deploy` task is run, it looks for an `.ftppass` file, which contains sets of usernames and passwords. When placing a Grunt environment inside a version control system, we must be wary of unauthorized access to login credentials. Therefore, it is good practice to place these credentials in an external file, which is not under version control. We could also use system environment variables to achieve the same effect.

Our `Gruntfile.js` above has set the `key` option to `"my-key"`, this tells `ftp-deploy` to look for this property inside our `.ftppass` file (which is in JSON format). So, we should create a `.ftppass` file like:

```
{
  "my-key": {
    "username": "john",
    "password": "smith"
  }
}
```

For testing purposes, there are free FTP servers available: PureFTPd `http://gswg.io#pureftpd` (Mac OS X) and FileZilla Server `http://gswg.io#filezilla-server` (Windows).

Once we have an FTP server ready, with the correct username and password, we are ready to transfer. Running this example should produce the following:

```
$ grunt
Running "ftp-deploy:target1" (ftp-deploy) task
>> New remote folder created /build/
>> Uploaded file: foo.js to: /
>> FTP upload done!
```

FTP is widespread and commonly supported; however, as technology and software improve, as legacy systems get deprecated, and as data encryption becomes a negligible computational cost, the use of unencrypted protocols like FTP is in decline—which segues us to SFTP.

SFTP

The Secure File Transfer Protocol is often incorrectly assumed to be a normal FTP connection tunneled through an SSH (Secure Shell) connection. However, SFTP is a new file transfer protocol (though it does use SSH).

In this example, we are copying three HTML files from our local `build` directory to the remote `tmp` directory. Again, to avoid placing credentials inside `build`, we store our `username` and `password` inside our `credentials.json` file. This example uses the Grunt plugin `http://gswg.io#grunt-ssh`. This plugin actually provides two tasks: `sftp` and `sshexec`, however, in this example we'll only be using the `sftp` task:

```
//Code example 10-sftp
module.exports = function(grunt) {

  // Load the plugin that provides the "sftp" task.
  grunt.loadNpmTasks('grunt-ssh');

  // Project configuration.
  grunt.initConfig({

    credentials: grunt.file.readJSON('credentials.json'),

    sftp: {
      options: {
        host: 'localhost',
```

```
        username: '<%= credentials.username %>',
        password: '<%= credentials.password %>',
        path: '/tmp/',
        srcBasePath: 'build/'
      },
      target1: {
        src: 'build/{foo,bar,bazz}.html'
      }
    }
  });

  // Define the default task
  grunt.registerTask('default', ['sftp']);
};
```

At the top of our configuration, we created a new `credentials` property to store the result of reading our `credentials.json` file. Using Grunt templates, which we cover in *Chapter 2*, *Setting Up Grunt*, we can list the path to the property we wish to substitute in. Once we have prepared our `credentials.json` file, we can execute grunt:

```
$ grunt
Running "sftp:target1" (sftp) task

Done, without errors.
```

We notice the `sftp` task didn't display any detailed information. However, if we run Grunt with the verbose flag: `grunt -v` we should see this snippet at the end of our output:

```
Connection :: connect
copying build/bar.html to /tmp/bar.html
copied build/bar.html to /tmp/bar.html
copying build/bazz.html to /tmp/bazz.html
copied build/bazz.html to /tmp/bazz.html
copying build/foo.html to /tmp/foo.html
copied build/foo.html to /tmp/foo.html
Connection :: end
Connection :: close

Done, without errors.
```

This output clearly conveys that we have indeed successfully copied our three HTML files from our local directory to the remote directory.

S3

Amazon Web Service's Simple Storage Service is not a deployment method (or protocol) like FTP and SFTP, but rather a service. Nevertheless, from a deployment perspective they are quite similar as they all require some configuration, including destination and authentication information.

Hosting Web Applications in the Amazon Cloud has grown quite popular in recent years. The relatively low prices of S3 make it a good choice for static file hosting, especially as running your own servers can introduce many unexpected costs. AWS has released a Node.js client library for many of its services. Since there was no Grunt plugins utilizing this library at the time, I decided to make one. So, in the following example, we are using `http://gswg.io#grunt-aws`. Below, we are attempting to upload all of the files inside the `build` directory into the root of the chosen bucket:

```
//Code example 11-aws
grunt.initConfig({
  aws: grunt.file.readJSON("credentials.json"),
  s3: {
    options: {
      accessKeyId: "<%= aws.accessKeyId %>",
      secretAccessKey: "<%= aws.secretAccessKey %>",
      bucket: "..."
    },
    //upload all files within build/ to output/
    build: {
      cwd: "build/",
      src: "**"
    }
  }
});
```

Again, similar to the SFTP, we are using an external `credentials.json` file to house our valuable information. So, before we can run this example, we first need to create a `credentials.json` file, which looks like:

```
{
  "accessKeyId": "AKIAIMK...",
  "secretAccessKey": "bt5ozy7nP9Fl9..."
}
```

Next, we set the `bucket` option to the name of bucket we wish to upload to, then we can go ahead and execute `grunt`:

```
$ grunt
Running "s3:build" (s3) task
Retrieving list of existing objects...
>> Put 'foo.html'
>> Put 'bar.js'
>> Put 2 files

Done, without errors.
```

Summary

In this chapter, we have learnt Grunt is an easy to use JavaScript build tool, which has the potential to greatly improve the development cycle of the typical front-end developer. We have covered many common build problems in this chapter and, by combining these examples, we see we can quite easily make use of various premade Grunt plugins to vastly simplify previously complex build processes.

In the next chapter, we will review the steps required to install Grunt and its only dependency—Node.js, and also the various methods of configuring Grunt.

Where to buy this book

You can buy Getting Started with Grunt: The JavaScript Task Runner from the Packt Publishing website: `http://www.packtpub.com/getting-started-with-grunt-the-javascript-task-runner/book`.

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

www.PacktPub.com

For More Information:
www.packtpub.com/getting-started-with-grunt-the-javascript-task-runner/book