Community Experience Distilled

Monitor and automate complex JavaScript tasks and processes by obtaining a practical understanding of Grunt

Learning Grunt

By the end of the book, you will have learned the usage of Grunt from scratch and be able to utilize it to the fullest.

Who this book is written for

If you are a JavaScript developer and want to learn project monitoring and automation using Grunt, then this book is for you. Basic knowledge of Node.js and Angular.js is assumed. However, no previous experience using Grunt.js is required.

What you will learn from this book

- Install and configure Grunt and its dependencies, such as Node.js and Node Package Manager (NPM)
- Create a sample application using Angular.js
- Configure plugins to perform various tasks that will benefit your project
- Explore the task-specific methods available through the Grunt API, such as configuration, externals, events, and logging
- Set up tasks to automate the build process of the simple contact application
- Build, test, and refine a sample project
- Get general tips, tricks, and troubleshooting strategies to identify and solve common issues in Grunt
- Advance your knowledge of Grunt with concepts such as configuration variables and test automation

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 4 'Configuration of Grunt Tasks'
- A synopsis of the book’s content
- More information on Learning Grunt
Douglas Reynolds is a web application developer. He holds a degree in software engineering and has worked with web technologies for over 15 years. Douglas currently works for Herff Jones as a Senior Web Applications Developer as a member of a highly skilled and dedicated team of professionals. Douglas writes and lectures on web technologies, teaches at Illinois State University in the Department of Technology, and writes on topics of technology for Packt Publishing. Douglas and his wife of over 20 years, along with their son and two dogs, live in Bloomington, Illinois. He loves his family and friends and enjoys spending time outdoors in nature, riding his Harley, and continually developing his relationship with God.
Preface

The essence of workflow automation is an exemplification of at least two of the three great virtues of a programmer.

*There are three great virtues of a programmer: laziness, impatience, and hubris.*


Larry Wall went on to explain that laziness is the quality that makes you go to great effort to reduce overall energy expenditure. Grunt provides task automation to relieve the developer of the mundane, repetitive processes that occur during software development. Expending effort up front with Grunt will pay dividends in reduction of energy expenditure in the development lifecycle. Impatience, Larry Wall said, makes you write programs that don't just react to your needs but actually anticipate them. Grunt automation can react to your needs by responding to events, such as file changing. With the right combination of tasks and configurations, Grunt automation can even seem as if it is anticipating your next step. *Learning Grunt* aims to provide you with a ramp from getting up and running to developing your own Grunt plugin. If you are at all lazy and impatient, in the context of a virtuous programmer, then *Learning Grunt* has been written just for you.

**What this book covers**

*Chapter 1, Grunt Dependencies*, will let us know that Grunt is a Javascript task runner, the purpose of which purpose is task automation. Before we can get started with Grunt, we need to make sure we understand and implement Grunt's prerequisites.

*Chapter 2, Foundation for Creating an App using Angular.JS*, will create a sample application to use throughout this book with the Angular-Seed scaffold.
Chapter 3, *All about Grunt Plugins*, introduces a sampling of common Grunt plugins, overview of their purposes, brief examples of implementation, and additional references and information.

Chapter 4, *Configuration of Grunt Tasks*, As we became familiar with installation and configuration of Grunt tasks. Focus is placed on two main files in a Grunt project; package.json and Gruntfile.js.

Chapter 5, *Task Setup in the sample_project Application*, defines the sample_project automated build process, introduces user stories, and then covers Gruntfile.js configuration from top to bottom.

Chapter 6, *Building the Sample Project*, will explain the process of loading tasks. We check functionality to ensure we get the expected results, and create custom and default tasks in order to automate the build process.

Chapter 7, *Advanced Grunt Concept*, will help you create your own custom task. Learn how to use the Grunt plugin tool and then write a custom Javascript plugin to Gzip files into a deployment directory.
Configuration of Grunt Tasks

In the previous chapter, an assortment of Grunt tasks were introduced, and information was provided to become familiar with how to obtain plugins as well as the syntax used to add plugins to your project. In this chapter, we will look at the actual installation and configuration of grunt tasks using the Angular Seed project. There will be two main files that we will be focusing closely on: package.json and gruntfile.js. These two files are where the grunt configuration takes place, some of it automatically, such as when we install a plugin and the package.json file is updated, as shown in the examples in Chapter 3, All about Grunt Plugins. Other configuration is completed manually so that plugin behavior can be defined- and customized as per the project's requirements.

- Reviewing the installation of Grunt tasks
- Discussing package.json and gruntfile.js
- Introducing the Grunt API

Reviewing the installation of Grunt tasks

In each task that was outlined in Chapter 3, All about Grunt Plugins there was a distinctly common thread that existed between all of the plugins: the installation process. Installation of plugins is accomplished through Terminal or command line, and the syntax is very simple. As previously described, here is the common format of the task installation syntax for grunt plugins:

```
npm install [package name] --save-dev
```
The `npm install` command invokes the installation of a package, `[package name]`. Additionally, this command has an optional flag, `-save-dev`, which instructs the installer to save the package as a development dependency. Plugins saved as devDependencies are saved locally to the project and registered in the package.json file in the devDependencies configuration section. The plugin itself, when installed as a devDependencies configuration, will be located in your project's node-modules directory. In this section, we will be installing plugins in the Angular Seed project (note that the installation order of plugins is not important).

### Installing contrib-jshint with NPM

As mentioned previously, one of the repetitive tasks that we will want to configure to run in our automated build will be linting our JavaScript. We will need the contrib-jshint plugin to perform this work for us. As has been shown, the installation process from plugin to plugin is virtually the same. We will perform the same process and have a look at the results of the installation process. Ensure that you are in the root of the sample_project directory:

```bash
cd [Your Project Directory]
```

As before, once confirmed that we are in the root of our sample_project directory, run the NPM install command for contrib-jshint:

```bash
npm install grunt-contrib-jshint --save-dev
```

Your output should be similar to the following contrib-jshint installation log. If you have a look at the output, you will see that some additional packages have been installed along with the plugin and all of the packages that we are installing. These additional packages are plugin dependencies that are required by the plugin. So, in the case of contrib-jshint, note that strip-json-comments, exit, shelljs, console-browserify, minimatch, cli, lodash, and htmlparser have all been installed as dependencies of jshint. You can find these additional packages in the npm directory, https://www.npmjs.com/, in order to learn more about each one. For instance, strip-json-comments allow you to use comments in your json files and then strip them out at build time. You can find information about strip-json-comments at https://www.npmjs.com/package/strip-json-comments:
Installation will automatically update our package.json file, registering contrib-jshint in devDependencies:

```
"devDependencies": {
  "bower": "^1.3.1",
  "grunt": "^0.4.5",
  "grunt-contrib-jshint": "^0.11.3",
  "grunt-contrib-watch": "^0.6.1",
  "http-server": "^0.6.1",
  "jasmine-core": "^2.3.4",
  "karma": "-0.12",
  "karma-chrome-launcher": "^0.1.12",
  "karma-firefox-launcher": "^0.1.6",
  "karma-jasmine": "^0.3.5",
  "karma-junit-reporter": "^0.2.2",
  "protractor": "^2.1.0",
  "shelljs": "^0.2.6"
}
```

As expected, contrib-jshint is installed in the node-modules directory located at the root of the sample_project directory:
This is all that needs to be done in order to install the contrib-jshint plugin in sample_project. The plugin is now ready to be configured in gruntfile.js; however, we will continue to install more plugins at this point. We will focus on configuration later on, without the interruption of installing plugins. It should be noted that it is perfectly fine to install plugins and configure, then later install more plugins and modify the gruntfile.js configuration. These steps can be repeated over and over to continually meet the requirements of your project. One of the beauties of Grunt is that it allows you to modify and scale your build requirements with your project as your project's needs evolve.

**Installing contrib-uglify with NPM**

As we progress through our build process, we will want to compress or minify our JavaScript so that we can decrease loading time and improve performance. The contrib-uglify plugin handles this compression with the optimization of JavaScript code, removal of unneeded whitespace, shortening of variable names, and other parsing operations, in order to reduce file size. The installation process is, again, the same as what we have been performing thus far. First, if you aren't already in your project directory, change directories to the root of your sample_project:

```
cd [Your Project Directory]
```

Once in the root of your sample_project, run the npm install command to install contrib-uglify in your project:

```
npm install grunt-contrib-uglify –save-dev
```

After running the install command, your console output should be similar to the following:

```
drcsoft-nbp:sample_project dourgrdotnet$ npm install grunt-contrib-uglify –save-dev
grunt-contrib-uglify@0.9.2 node_modules/grunt-contrib-uglify
    – uri-path@0.0.2
    – chalk@1.1.1 (supports-color@2.0.0, escape-string-regexp@1.0.3, ansi-styles@2.1.0, strip-ansi@3.0.0, has-ansi@2.0.0)
    – uglify-js@2.4.24 (uglify-to-browserify@1.0.2, async@0.2.10, yargs@3.5.4, source-map@0.1.34)
    – lodash@3.10.1
    – maximin@1.1.0 (figures@1.4.0, gzip-size@1.0.0, pretty-bytes@1.0.4)
drcsoft-nbp:sample_project dourgrdotnet$
```
As we would expect, package.json will have been updated to add the registration of contrib-uglify to the devDependencies section:

```
"devDependencies": {
  "bower": "^1.3.1",
  "grunt": "^0.4.5",
  "grunt-contrib-jshint": "^0.11.3",
  "grunt-contrib-uglify": "^0.9.2",
  " grunt-contrib-watch": "^0.6.1",
  " http-server": "^0.6.1",
  " jasmine-core": "^2.3.4",
  " karma": "-0.12",
  " karma-chrome-launcher": "^0.1.12",
  " karma-firefox-launcher": "^0.1.6",
  " karma-jasmine": "^0.3.5",
  " karma-junit-reporter": "^0.2.2",
  " protractor": "^2.1.0",
  " shelljs": "^0.2.6"
}
```

A quick inspection of our sample_project's node-modules directory should confirm that the contrib-uglify package has been successfully installed:
Configuration of Grunt Tasks

Installing contrib-less with NPM

The sample_project will illustrate the use of LESS, which provides extensibility to CSS through the use of variables, mixins, and conditional logic. The contrib-less task handles the compilation of LESS files into CSS. An additional bonus is that we can use contrib-less to handle our CSS minification to help reduce file size. As we have done in each package installation, ensure that you are in the root of your sample_project directory, if not, change directories to the project root:

cd [Your Project Directory]

Once confirmed that you are in the root of your project, run the npm install command in order to install contrib-less in sample_project:

`npm install grunt-contrib-less -save-dev`

When the install command is completed, you should see a console output similar to the following:

```
drcsoft-mbp:sample_project dougrdotnet$ npm install grunt-contrib-less -save-dev
grunt-contrib-less@1.0.1 node_modules/grunt-contrib-less
  async@0.9.2
  lodash@2.4.2
  chalk@0.5.1 (ansi-styles@1.1.0, escape-string-regexp@1.0.3, supports-color@0.2.0, strip-ansi@0.3.0, has-ansi@0.1.0)
  less@2.4.0 (graceful-fs@0.0.8, mime@1.3.4, image-size@0.3.5, mkdirp@0.5.1, source-map@0.2.0, errno@0.1.4, promise@0.11.1, request@2.64.0)
drcsoft-mbp:sample_project dougrdotnet$
```

Note that package.json has been updated with the contrib-less registration:

```
"devDependencies": {
  "bower": "^1.3.1",
  "grunt": "^0.4.5",
  "grunt-contrib-jshint": "^0.11.3",
  "grunt-contrib-less": "^1.0.1",
  "grunt-contrib-uglify": "^0.9.2",
  "grunt-contrib-watch": "^0.6.1",
  "http-server": "^0.6.1",
  "jasmine-core": "^2.3.4",
  "karma": "^-0.12",
  "karma-chrome-launcher": "^0.1.12",
  "karma-firefox-launcher": "^0.1.6",
  "karma-jasmine": "^0.3.5",
```

[70]
The contrib-less package should now be installed in the sample_project's node-modules directory:

```
"karma-junit-reporter": "^0.2.2",
"protractor": "^2.1.0",
"shelljs": "^0.2.6"
```

### Installing contrib-watch with NPM

The first plugin that we will install is the contrib-watch task. This detects file changes and responds to these events by running additional configured tasks. The configuration takes place in gruntfile.js. To put this in a clearer perspective, our project will have JavaScript files that be modified frequently. Every time we save modifications, we want to run jshint on the files to lint them for errors. Rather than having to do this manually each time, we will configure contrib-watch to automate the linting process for us. More on this configuration later; for now, let's just get contrib-watch installed.

Open a command prompt and change directories to your Angular Seed sample project root directory that we created in Chapter 2, Foundation for Creating an app using Angular.JS:

```
  cd [Your Project Directory]
```
Once we are in our project directory, we can simply install contrib-watch with the following npm command:

```
npm install grunt-contrib-watch --save-dev
```

Your output should look similar to the following:

```
$ npm install grunt-contrib-watch --save-dev

devDependencies:

```
"devDependencies": {
  "bower": "1.3.1",
  "grunt": "0.4.5",
  "grunt-contrib-watch": "0.6.1",
  "http-server": "0.6.1",
  "jasmine-core": "2.3.4",
  "karma": "-0.12",
  "karma-chrome-launcher": "0.1.12",
  "karma-firefox-launcher": "0.1.6",
```
"karma-jasmine": "^0.3.5",
"karma-junit-reporter": "^0.2.2",
"protractor": "^2.1.0",
"shelljs": "^0.2.6"
}

The contrib-watch plugin is now installed in the node-modules directory located at the root of the sample_project directory:

```

The contrib-watch plugin is now installed in sample_project and ready to be configured in gruntfile.js. Before moving on to configuration, let's continue to install additional plugins that we will need to get going.

### Discussing package.json and gruntfile.js

Grunt projects have two important files that are required in order to define and configure Grunt: package.json and gruntfile.js. The package.json is a JSON file that is normally located at the root of a project using NPM to manage dependencies, and gruntfile.js is a JavaScript file that also exists in the project root directory. The combination of these two files provides the information about the project and configuration of the packages used in the project that are needed in order to create an automated build system. We will look at each in more detail before beginning to configure sample_project.
The package.json file

As mentioned, package.json is a JSON file that defines data about our project, also known as project metadata. In this file, as we have seen throughout our plugin installation process, we have registered our project's plugins in the devDependencies section.

The metadata that package.json defines, in addition to the project dependencies, is a set of properties that describes your project. Name and version are both required fields:

- **Name**: This is the name of the application. It must be shorter than 214 characters, must not start with a dot or underscore, and must not contain any non-URL-safe characters.
- **Version**: This is the version of your application. It must be able to be parsed by node-semver (https://github.com/npm/node-semver). The combination of the name and version is required because, together, they will create a unique application identifier.

In sample_project, there are some additional fields added during our Angular Seed project creation, which include private, description, repository, and licence:

- Private: This instructs npm to never publish your project and prevents the possibility of accidentally publishing a private repository
- Description: This is a brief excerpt that describes the project
- Repository: This is the location of the project code
- Licence: This defines how the application can be used by others

We have been watching the devDependencies section of the package.json develop as we have worked through the package installation process. The devDependencies is really just a simple JSON object whose properties define the package dependencies of our project. Notice each entry in devDependencies as it exists at this time:

```
"devDependencies": {
    "bower": "^1.3.1",
    "grunt": "^0.4.5",
    "grunt-contrib-jshint": "^0.11.3",
    "grunt-contrib-less": "^1.0.1",
    "grunt-contrib-uglify": "^0.9.2",
    "grunt-contrib-watch": "^0.6.1",
    "http-server": "^0.6.1",
    "jasmine-core": "^2.3.4",
    "karma": "^0.12",
```
Each line defined in the `devDependencies` object is a map and key-value pair that matches up an application name with a version. Both the name and version are strings and the version may include a descriptor. If the version has no descriptor, then the version string must be matched exactly. If the > version descriptor is used, then the version must match any version greater than the value. Other descriptors include the following:

- `>=`: The version must be greater than or equal to the value
- `<`: The version must be less than the value
- `<=`: The version must be less than or equal to the value
- `~`: The version must be the approximate equivalent to the value
- `^`: The version must be compatible with the value
- `1.2.x`: The version may match any minor version, for example, `1.2.8`
- `*`: The version may match any versions

Note that the package.json file also contains a `scripts` dictionary object. This object defines scripts that should be run as part of the dependency and when the script should be run. Let's look at our `scripts` object as an example:

```json
"scripts": {
  "postinstall": "bower install",
  "prestart": "npm install",
  "start": "http-server -a localhost -p 8000 -c-1",
  "pretest": "npm install",
  "test": "karma start karma.conf.js",
  "test-single-run": "karma start karma.conf.js --single-run",
  "preupdate-webdriver": "npm install",
  "update-webdriver": "webdriver-manager update",
  "preprotractor": "npm run update-webdriver",
  "protractor": "protractor e2e-tests/protractor.conf.js"...
truncated for brevity
}
```
Configuration of Grunt Tasks

Notice that each object maps a command to a script. For example, the postinstall command is run after the package is installed, prestart is run by the npm start command, start is also run by the npm start command, and pretest, test, and test-single-run are all run by the npm test command. More information on the scripts dictionary can be found in the npm documentation at https://docs.npmjs.com/misc/scripts.

The gruntfile.js file

Up to this point, we have been putting off gruntfile.js as something that we will get into later. Finally, it is time to discuss gruntfile in detail. As mentioned previously, gruntfile, for the purposes of sample_project, is a valid JavaScript file. Note that it may also be written in CoffeeScript. Gruntfile should be located at the root of the project and part of your project source code.

A gruntfile will include a wrapper function that will contain all of the grunt-related code within. The wrapper function syntax looks as follows:

```javascript
module.exports = function(grunt) {
  // grunt related code
};
```

The module.exports encapsulates the code within to create a single module of code. The exports object becomes the result returned when a require call is made, and module.exports is set equal to a function that takes a grunt object as its argument. The wrapper function will contain the following parts:

- Project and task configuration
- Loading Grunt plugins and tasks
- Custom tasks

In the wrapper method, there will be a call to the grunt.initConfig method. Most of the grunt plugins will use initConfig for configuration. In sample_project, one of the first items to be configured is getting the project's metadata from package.json so that we can use it in our automated build. We will get the name, version, and current date so that we can create a header banner that can be used in our build files. Here is a code example that we can walk through and discuss:

```javascript
module.exports = function(grunt) {
  grunt.initConfig({
    pkg: grunt.file.readJSON('package.json'),
    banner: '/*! <%= pkg.name %> */
    uglify: {
      ...}
```
options: {
    banner: '<%= banner %>'
},

...{
     
}});

The **module.exports** object contains all of our grunt-related code. The first section of our configuration is the **grunt.initConfig** method. In **initConfig**, we created an object property named **pkg** and assigned it to the value of the metadata contained in package.json. We then created a banner property that creates a comment string. This will be used as a banner comment in the minified JavaScript created by contrib-uglify. The banner property string uses template string variables identified by `<% %>`, which will be replaced with content from package.json. The `<%= pkg.name %>` template string variable gets the value of the name property from package.json and replaces the variable with this content. When the template string variable is replaced, it will read angular-seed, which is the name of the sample_project project.

Now, we get to take a sneak peak of the plugin configuration in order to show how we can use the banner in the configuration of contrib-uglify. In the following abbreviated contrib-uglify configuration, we simply need to add the banner to the configuration options:

```javascript
uglify: {
    options: {
        banner: '<%= banner %>'
    },

    ...
}
```

The **uglify** method takes an optional banner configuration so that a comment header may be included in the minified output files with which it creates. Let's have a the banner example:

```
banner: '/**! <%= pkg.name %>' + ' <%= grunt.template.today("yyyy-mm-dd") %>' + '
```

The following comment will be created and added to the head of the generated minified JavaScript. The result is a template string variable replacement and concatenation of the name and date, obtained from the **grunt.today** method:

```
/**! angular-seed 2015-09-28 */
```
Moving beyond task configuration and into loading Grunt plugins, we need to understand the prerequisites of loading a plugin. All of the work that we have done thus far in installing plugins has created our plugin mappings in `devDependencies`. The plugin will be installed in the node-modules directory using `npm install`. Once the plugin exists in node-modules and is mapped in `devDependencies`, then it may be loaded in `gruntfile.js`. Loading, or enabling, a plugin is very simple and only takes a single line of code. The following is the code needed in order to load contrib-uglify:

```
grunt.loadNpmTasks('grunt-contrib-uglify');
```

Loading plugins takes place below the `grunt.initConfig` method, following its closing brace. For instance, loading contrib-uglify would look something like this:

```javascript
module.exports = function(grunt) {
  grunt.initConfig({
    pkg: grunt.file.readJSON('package.json'),
    ...
  });
  grunt.loadNpmTasks('grunt-contrib-uglify');
};
```

In Grunt, you can define a task to run as the default task. In fact, you can define multiple tasks to run as the default task. This is known as creating a custom task. When you define a custom default task, it will always run if nothing else is defined. Recall that each task had its own command that could be used to run the task from the command line manually. The contrib-uglify command can be run manually by issuing the `grunt uglify` command. Notice that we use the `grunt` command and then specify the task to be run. If we were to run the `grunt` command by itself, with a default task defined, it would run automatically. We can create a default command that controls the order with which our plugins run and only have to issue the `grunt` command to invoke our build process, if we so desire.

Let’s take a look at how we create a custom task. First, let’s look at the scenario that we just outlined and create a custom task that will automatically run contrib-uglify as the default task. The syntax to create a custom task is also simple. Here is our default task example:

```
grunt.registerTask('default', ['uglify']);
```
Here, we simply register the task, define it as default, and provide an array of tasks to be run. In this case, our array only has a single item. Did you catch that? As the registerTask method takes an array of tasks as an argument, we can define multiple tasks to run as our default task. What if we wanted to lint our JavaScript, then minify the linted scripts, and then compile our LESS into CSS, all in one build command? This would be very simple using the registerTask method:

```javascript
grunt.registerTask('default', ['jshint', 'uglify', 'less']);
```

Custom tasks aren't just limited to grunt plugins. Custom JavaScript can be used in a custom task. Javascript methods can be included in the registerTask method as follows:

```javascript
grunt.registerTask('default', 'myTaskName', function() {
  //some javascript here
});
```

It is also possible to load external JavaScript files using the grunt.loadTasks method. The syntax to use this method is as follows:

```javascript
grunt.task.loadTasks(pathToFile)
```

You can also use the following method:

```javascript
grunt.loadTasks(pathToFile)
```

Grunt provides a sample gruntfile that can be used to get started quickly. While the sample may not be exactly what is needed, it is a good way to get started quickly with a configuration file that can be easily modified to meet project needs. To get started, we can get the sample gruntfile.js from the source:

```javascript
module.exports = function(grunt) {

  grunt.initConfig({
    pkg: grunt.file.readJSON('package.json'),
    concat: {
      options: {
        separator: ';'
      },
      dist: {
        src: ['src/**/*.js'],
        dest: 'dist/<%= pkg.name %>.js'
      }
    },
    uglify: {
      options: {
```
Configuration of Grunt Tasks

```javascript

    banner: '/*! <%= pkg.name %> <%= grunt.template.today("dd-mm- yyyy") %> */
    },
    dist: {
        files: {
            'dist/<%= pkg.name %>.min.js': ['<%= concat.dist.dest %>']
        }
    },
    qunit: {
        files: ['test/**/*.html']
    },
    jshint: {
        files: ['Gruntfile.js', 'src/**/*.js', 'test/**/*.js'],
        options: {
            // options here to override JSHint defaults
            globals: {
                jQuery: true,
                console: true,
                module: true,
                document: true
            }
        }
    },
    watch: {
        files: ['<%= jshint.files %>'],
        tasks: ['jshint', 'qunit']
    });

grunt.loadNpmTasks('grunt-contrib-uglify');
grunt.loadNpmTasks('grunt-contrib-jshint');
grunt.loadNpmTasks('grunt-contrib-qunit');
grunt.loadNpmTasks('grunt-contrib-watch');
grunt.loadNpmTasks('grunt-contrib-concat');
grunt.registerTask('test', ['jshint', 'qunit']);
grunt.registerTask('default', ['jshint', 'qunit', 'concat', 'uglify']);
```
Hopefully, the amount of content in the file isn't of concern; each section has already been discussed. Looking at the sample gruntfile.js, let's break it down section by section. Notice that everything in gruntfile.js is enclosed in the wrapper. At the top of the document, we have our `grunt.initConfig` method where all of our plugin configuration takes place.

The contents of package.json are imported to the `pkg` property so that we can get at the values defined in package.json, such as name, description, and version.

This configuration uses `grunt-contrib-concat`. The purpose of `contrib-concat` is to concatenate multiple files into a single file. It is important to be sure to use the correct separator when concatenating JavaScript files, so the options configuration defines a semicolon as the separator option. The `dist` options defines the source files to be concatenated and the destination files where the concatenated output will be written. Then, we have a closing brace and comma, which leads us to our next plugin configuration.

Next, contrib-uglify is configured to use the optional banner so that it will automatically include header comments in the minified output file. The `dist` option defines the filenames for source and destination and also uses template string variables for dynamic file naming within the files configuration:

```javascript
files: {
  'dist/<%= pkg.name %>.min.js': ['<%= concat.dist.dest %>']
}
```

The `files` parameter defines the path to source, such as `dist/angular-seed.min.js`, and concatenates it with the file that was produced from the concat plugin (`concat.dist.dest`).

The sample gruntfile configures the contrib-qunit task that simply runs the unit test files. It takes a files parameter to define the path to the test runner; the asterisks are path wildcards:

```javascript
files: ['test_runner/**/*.html']
```

Following qunit, we find the contrib-jshint configuration. The files parameter defines the files that will be linted:

```javascript
files: ['gruntfile.js', 'src/**/*.js', 'test/**/*.js']
```

Notice that gruntfile.js is in the list of files to lint. This is very good practice and will help ensure that no JavaScript errors are introduced into the gruntfile itself. In the preceding example code, the files to be linted include gruntfile.js, all .js files under the src directory, and all .js files under the test directory.
Configuration of Grunt Tasks

The last plugin to configure in the sample gruntfile is the watch plugin. The contrib-watch plugin responds to changes in files defined by you and runs additional tasks upon being triggered by the changed file events. In this, we define the files to watch and the tasks to run if a watched file changes:

```javascript
files: ['<%= jshint.files %>'],
tasks: ['jshint', 'qunit']
```

In this case, the files that are defined by the contrib-jshint plugin files parameter will be watched and the contrib-jshint and contrib-qunit tasks will be run if any of these watched files change.

Now that all of the configuration is complete for the sample gruntfile plugins, the next step is to load all of the plugins that we will need. This is done in the lines using the grunt.loadNpmTasks method. This is very straightforward and self-explanatory:

```javascript
grunt.loadNpmTasks('grunt-contrib-uglify');
grunt.loadNpmTasks('grunt-contrib-jshint');
grunt.loadNpmTasks('grunt-contrib-qunit');
grunt.loadNpmTasks('grunt-contrib-watch');
grunt.loadNpmTasks('grunt-contrib-concat');
```

The final step in the sample gruntfile is to define the custom tasks. The first custom task is set up to run the unit tests. The task is named ‘test’ and will run the jshint and qunit tasks using the grunt test command:

```javascript
grunt.registerTask('test', ['jshint', 'qunit']);
```

Finally, the second custom task defines the default task for the gruntfile. This is the task that will define the actual automated build process. In the default task, the name is defined as default and the tasks that will be run are contrib-jshint, contrib-qunit, contrib-concat, and contrib-uglify. It is important to note that these tasks will be run in the order that they are added to the array of tasks argument to the registerTask method:

```javascript
grunt.registerTask('default', ['jshint', 'qunit', 'concat', 'uglify']);
```

The default task can be run using the grunt command or custom task name in the grunt.default command.
Introducing the Grunt API

Uptil now, we haven't discussed any details of the Grunt Application Programming Interface (API). If the appearance of grunt in or around all of the gruntfile code snippets that we have seen so far has caught your eye, then you are to be congratulated for noticing that grunt is providing some things that we need in order to set up and configure our automated tasks.

Grunt exposes all of its properties and methods via the grunt object. Returning to the wrapper function that was provided as an example in the beginning of the gruntfile.js section, we can see that its anonymous function takes the grunt object as its only argument:

```javascript
module.exports = function(grunt) {
    // grunt related code
};
```

It is through the grunt object that we can access all of its properties and methods and this is why this wrapper method is required.

A discussion of modules will be useful in order to get a better understanding of how node.js can share items, such as objects, properties, and methods, between files. The `module.exports` is a node.js pattern that allows the code within the function to be encapsulated into a single module of code. This module then can expose its contents in order to export whatever is asked for from the caller. The caller uses `require` to import a module. So what problem does this solve? In Node.js, declarations made within a file have scope to only that file. In order to provide a means to share these items across files, we can create modules that expose their contents and allow us to import them to other files. Imagine that we have an external JavaScript file, `sum.js`, and within it, we have a function that returns the sum of two numbers provided as arguments:

```javascript
//sum.js
var sumOfTwoNumbers = function (x, y) {
    return x + y;
}
```

Only code within `sum.js` can call the `sumOfTwoNumbers` method. However, if we create a module, then Node can map from one file directly to another file. This is accomplished with the `require` method. The purpose of `require()` is to load, or import, a module to the calling file. So, for instance, we could load the `sum.js` file to a file named `calculate.js` with the following line of code:

```javascript
//calculate.js
var myModule = require('./sum');
```
Configuration of Grunt Tasks

This would load our sum.js file as a module; however, sum.js still doesn't expose anything because we need to use the module.exports function to expose the contents. In order to do this, we can modify sum.js as follows:

```javascript
//sum.js
var sumOfTwoNumbers = function (x, y) {
  return x + y;
}
module.exports.sumOfTwoNumbers = sumOfTwoNumbers;
```

At this point, we now have a sum module that exposes its `sumOfTwoNumbers` method. We can now use this module to calculate a sum for us in the following manner:

```javascript
//calculate.js
var myModule = require('./sum');
var sum = myModule.sumOfTwoNumbers(5, 7);
console.log('The sum of 5 and 7 is: ' + sum);
```

We can test our small node.js application very easily. Create the two files, sum.js and calculate.js, and place them temporarily in the root of the project site. Using the code from the examples, you should have a sum.js file that looks as follows:

```javascript
//sum.js
var sumOfTwoNumbers = function (x, y) {
  return x + y;
}
module.exports.sumOfTwoNumbers = sumOfTwoNumbers;
```

You should also have a calculate.js file that looks like the following:

```javascript
//calculate.js
var myModule = require('./sum');
var sum = myModule.sumOfTwoNumbers(5, 7);
console.log('The sum of 5 and 7 is: ' + sum);
```

Ensure that you are in the root of the sample_project directory:

`cd [Your Project Directory]`

Now, issue the following command from Terminal in order to run our small node.js program:

`node calculate.js`
The output in Terminal should be the content of our `console.log()` method in `calculate.js`:

```
# sample_project — bash — 84×29

drcsoft-nb:sample_project douchrdotnet$ node calculate.js
The sum of 5 and 7 is: 12

drcsoft-nb:sample_project douchrdotnet$  
```

There are other ways to set up `module.exports`; however, the point here is simply to introduce the concept of using `require` to import modules and share code across multiple files. For ECMAScript 6 implementation of modules and module syntax, see the ES6 Module Syntax page at https://github.com/ModuleLoader/es6-module-loader/wiki/Brief-Overview-of-ES6-Module-syntax.

Referring to our `gruntfile.js`, we can now look at the wrapper function with the new understanding that `module.exports` is creating a module of our configuration files. The anonymous function that `module.exports` is set equal to accepts a parameter, `grunt`. The grunt argument is the `grunt` object whose properties and methods we are using in our configuration, task loading, task registration, and custom task creation. Here are some examples of how we have used the `grunt` object in `gruntfile.js` thus far.

### The wrapper function and `initConfig` Method

We will have a better look as wrapper function in this section:

```javascript
module.exports = function(grunt) {
    grunt.initConfig({
        pkg: grunt.file.readJSON('package.json'),
        banner: '/**! <%= pkg.name %> */
    ,
        uglify: {
            options: {
                banner: '<%= banner %>'
            },
            ...
        }
    });
};
```
Configuration of Grunt Tasks

The `initConfig` method is an alias for the `grunt.config.init` method. The purpose of `grunt.initConfig` is to initialize the grunt configuration object. Within the configuration, we begin with creating a `pkg` object with a value of the contents of package.json. Grunt exposes the `grunt.file` object's `readJSON` method that takes a parameter of a valid JSON file. The `grunt.template.today` method is a helper function that can be used to obtain and format today's date. Notice the format string that it takes as an argument. The `yyyy-mm-dd` is a format string that will output today's date in the format of the full year, two-digit month, and two-digit date of month.

**Loading NPM tasks:**

Let's see how to load NPM tasks:

```javascript
grunt.loadNpmTasks('grunt-contrib-uglify');
grunt.loadNpmTasks('grunt-contrib-jshint');
grunt.loadNpmTasks('grunt-contrib-qunit');
grunt.loadNpmTasks('grunt-contrib-watch');
grunt.loadNpmTasks('grunt-contrib-concat');
```

The `grunt.loadNpmTasks` method takes an argument of a string name that specifies a locally installed plugin that was installed through NPM.

**Creating tasks**

The `grunt.registerTask` method registers an alias, `'test'` in this example, with a task list of one or more tasks.

```javascript
grunt.registerTask('test', ['jshint', 'qunit']);
```

**Grunt fail API**

The `grunt.warn` method will display an error and abort Grunt. There is a flag that can be used in order to warn and continue processing tasks, the `-force` flag. The `grunt.fatal` method will also display an error and abort Grunt.

**The Grunt event API**

Grunt itself does not dispatch any events; however, the Grunt event API provides the ability to define event listeners and handler methods. Some event methods include the following methods:

- `grunt.event.on(event, listener)`: This adds a listener to the array of listeners for the event defined in the method
grunt.event.once(event, listener): This adds a one-time-use listener for the specified event, after which time that the event is dispatched, the listener is removed from the array of listeners

grunt.event.off(event, listener): This method removes an event listener from the array of listeners

grunt.event.removeAllListeners([event]): This removes all listeners from the array of listeners

The Grunt file API

The file API provides methods for file operations, such as reading and writing files and finding files within the filesystem. Grunt leverages many node.js file methods, adding additional options such as logging and error handling.

Reading and writing files are the more common uses of the Grunt File API. The grunt.file.read, by default, returns a string representation of the file's contents. In the Grunt template that was used as an example earlier in this chapter, we saw a method, grunt.file.readJSON. Like the read method, readJSON reads a file and returns a result of JSON formatted data. Another read method is grunt.file.readYAML, which does as expected, and returns YAML formatted data.

Grunt provides methods for write operations, including grunt.file.write, to write content to files and create directories based on the filepath parameter. The write method has a --no-write flag that will prevent write from actually writing the file. This is useful for a test run to ensure that the file writing task behaves as expected. File copy processing is handled by grunt.file.copy. The copy method takes a source, destination, and option as its arguments and will create directories as needed. Like file.write, file.copy also has a --no-write flag.

Deleting files with recursive deletion of directories is accomplished with the grunt.file.delete method. The file.delete method will delete the file path specified in its arguments without deleting the current working directory. However, file.delete has a --force option that will allow the file.delete operation to delete the current directory as well. The file.delete method also has a --no-write flag.
The directories methods in grunt provide two methods—grunt.file.mkdir creates directories along with any directories needed to be created as provided in the dirpath parameter; file.mkdir command provides a mode parameter that is used to specify directory permissions. If no mode is specified, then the file permissions on a newly created directory will default to 0777 (rwxrwxrwx). The second method is grunt.file.recurse; file.recurse allows recursively accessing directories and executing a callback function for each file contained within. The callback receives the absolute path, root directory, subdirectory, and filename as arguments that can then be used in the callback.

The Grunt file API also includes methods for filename expansion (globbing). Some examples that use expansion are grunt.file.expand, which returns an array of file paths or an array of directory paths. The grunt.file.expandMapping method returns an array of src-dest file mapping objects. The grunt.file.match method returns an array of file paths that match a specified globbing pattern. See the NPM glob project for additional information at https://www.npmjs.com/package/glob. Like the file.match method, grunt.file.isFileMatch matches files to patterns; however, file.isFileMatch method returns a Boolean—true, if matching file(s) are found and false, if not.

Additionally, the API provides methods for file type operations such as grunt.file.exists, which checks for the existence of a file, returning a Boolean of a given path result. In order to check whether a given path is a symbolic link, grunt.file.isFileLink will return a Boolean result. To check for the existence of a directory, grunt.file.isDirectory will return a Boolean given a specified path. Similarly, in order to determine if a specified path is a file, grunt.file.isFile will provide a Boolean result.

The last section of the Grunt file API is methods for path operations. The grunt.file.isDirectoryAbsolute method returns a Boolean result of whether the given path is an absolute path. To check whether more than one path refers to the same path, grunt.file.arePathsEquivalent will return a Boolean. The grunt.file.doesPathContain method returns a Boolean whether a path's descendants are contained within a specified ancestor path.
The Grunt log API

The main purpose of the grunt.log API is to output messages to the console. The grunt.log.write flag creates a log of a string taken as a msg argument. If you wish to write a log with line breaks, grunt.log.writeLn adds a trailing new line character to the end of the msg argument. In order to log a msg as an error, the grunt.log.error method should be used. If the msg is null, red ERROR text will be logged; otherwise, the error message will be logged with a new line character added to its end. Formatting an error message to 80 columns of text is achieved with the grunt.log.errorLns method. To log an OK message in green, use the grunt.log.ok method with a null msg argument; otherwise, it will log the message with a trailing new line character. To wrap an OK message to an 80-column format, use the grunt.log.okLns method. Bold text can be added to a log message using grunt.log.subhead. This will also add a new line character to the end of the message. Logging object properties can be accomplished with the grunt.log.writeFlags method. The log.writeFlags is useful for debugging, as is grunt.log.debug, which logs a debug message only if the --debug flag is used.

The Grunt log API has some utility methods that provide a string result that can be used by other methods, not actually generating log entries themselves. For instance, grunt.log.wordlist returns an array of comma-separated items. The separator can be defined in options and can also take a color option to color the separator. In order to remove all color from a result, grunt.log.uncolor will remove all color information from a string. In order to wrap text at a certain number of characters, the grunt.log.wraptext method should be used. Should a table format of text-aligned to columns be desired, grunt.log.table will generate output in columns.

The Grunt option API

We have actually seen the Grunt options API being used already. In the configuration section of the sample gruntfile.js earlier, we saw the options being defined for the tasks that we were configuring. For instance, in our pseudo-configuration we saw the following use of options:

```javascript
uglify: {
  options: {
    banner: '<%= banner %>'
  },
  ...{
  }
}
```
The options API provides tasks with a means to share parameters and access command-line flags, such as the --debug flag that was mentioned with the grunt.log.debug method. The grunt.option simply gets or sets an option. It takes a key, such as debug, and a value, such as true, which can then be used to determine if the current task should be run in debug mode. The grunt.option.init initializes an option and grunt.option.flags returns an array of command-line parameters.

Grunt Template API

In the sample gruntfile.js found earlier in this chapter, we saw some examples of templates being used. Recall that we loaded the contents of package.json in an object named pkg and then were able to refer to the properties in the pkg object through dot notation:

```javascript
dest: 'dist/<%= pkg.name %>.js'
```

In this snippet, we are referring to the name property of the pkg object with pkg.name. Notice the <%= and %> that surround pkg.name. These characters are template delimiters and allow the string representation of the value of the template within to be expanded when it is used. The grunt.template.process processes Lo-Dash template strings. Discussion of Lo-Dash is out of the scope of this book's purpose; more information on Lo-Dash can be found at https://lodash.com/docs/#template. The delimiters shown are the default template delimiters; however, these can be changed with template.process using options.delimiters to set a custom delimiter. The template.process uses grunt.template.setDelimiters internally to set delimiters, and template.setDelimiters can be used on its own to set custom delimiters too, but it is most commonly achieved with template.process.

It should be noted that the grunt object is exposed within a template, so all of the grunt properties and methods are available inside the template delimiters. A simple example would be setting the formatted date using the grunt.template.today method as we saw previously when creating a banner string that was used in the contrib-uglify configuration:

```javascript
banner: '/*! <%= pkg.name %> ' + ' <%= grunt.template.today("yyyy-mm-dd") %>}' + ' 
''
```

The Grunt task API

The last Grunt API that we will cover is the Grunt task API. We have already seen the API in use in our sample gruntfile earlier in this chapter. The three main purposes of the Grunt task API are to register, run, and load external tasks.
The grunt.task.registerTask registers either an alias task or task function. If

task.registerTask is being used to register an alias task, task.registerTask

creates an alias for one or more tasks contained within an array of tasks. In the case

that a description and function are provided as arguments to task.registerTask,

the function argument will be executed when the task is run. The grunt.task.

registerMultiTask registers what is known as a multi-task. A multi-task is a task

that will run all of its named targets if no specified target exists, or if the target is

specified, then it will run only that particular target.

The grunt.task.require ensures that a required task successfully completes;

otherwise, the task that requires it will be failed. This allows a task dependency to

be created. The grunt.task.exists checks for the existence of a task and returns

a Boolean based on the result. The grunt.task.renameTask provides the ability to

rename a task and perhaps override the old task with new behavior in the new task.

External tasks are loaded from a specified directory with grunt.task.loadTasks.

Similarly, grunt.task.loadNpmTasks loads external tasks that were installed with

NPM. Tasks can be queued and run on command. The grunt.task.run will run an

array of tasks independently from Grunt running all tasks from the command line.

The grunt.task.clearQueue will empty any tasks that are queued, entirely.

Summary

In this chapter, we looked at the installation and configuration of Grunt tasks. Focus

was placed on two main files in a Grunt project: package.json and Gruntfile.js. As

we discovered, the setup process was recorded in package.json as we installed

our tasks. You learned about properties and objects that are contained in package.

json and discussed the purpose of these properties. The majority of configuration is

accomplished in Gruntfile.js. In this file, we encapsulated everything in a wrapper

function so that we could export the contents of the file as a module. The main

sections of Gruntfile.js consisted of the wrapper function, configuration, task loading,

and task registration/custom task creation. After the discussion of Gruntfile.js

details, we went deeper into the Grunt API. In this section, we discussed what the

Grunt API is and covered specific methods of the API for different types of purposes,

for instance, the log API and task API.

In Chapter 5, Task Setup in the sample_project Application we will jump into the actual

configuration and building of sample_project. Details about the process of running

automated builds and inspecting the results of the build process will be covered

as we progress. We will take what has been learned about the setting up and

configuration of tasks in this chapter, and apply this knowledge to the process of

building, testing, and refinement.
Where to buy this book

You can buy Learning Grunt from the Packt Publishing website.

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

Click here for ordering and shipping details.