Power-Up Your **Front-End** Development With **Grunt**

Belén Albeza

Power-up Your Front-End Development with Grunt

A Quick Start Guide to Grunt

Belén Albeza

This book is for sale at http://leanpub.com/grunt

This version was published on 2013-07-30

Contents

About

This book is a practical guide that shows how to automate front-end projects using Grunt.

My primary goal is to reach as many developers as possible, that's why this book is released **DRM-free** and under a **Creative Commons** license that allows re-distribution for free.

Share this book!

Please share this book with your friends and peers, as well as in BitTorrent networks.

If you have received this book from a friend, or you have downloaded it, please consider **purchasing it** if you find it useful. You can buy this book at http://www.leanpub.com/grunt[1].

Last, you can download all the sample source code from http://www.belenalbeza.com/books/grunt[2]

Thank you, and happy coding!

Belén

[1] http://www.leanpub.com/grunt
[2] http://www.belenalbeza.com/books/grunt

Hello Grunt

This book shows how to **automatise common tasks** in front-end development, like running tests or compiling Sass stylesheets. While we could do this with shell scripts, we will use a tool specially tuned for JavaScript development: Grunt[3]

Grunt is a **task runner**. We could define a task that starts a local web server and just launch it from the console with:

```
$ grunt server
```

Okay, running a web server is not very impressive. But what if we could, with that very same line, lint our code, compile Sass stylesheets, Handlebar templates, minify all the JavaScript files, run unit tests and *then* launching a server and automatically open a browser with your web app on it?

That sounds better, and these kinds of flows is what Grunt is made for.

This book is purely practical, and we will see how to automatise some of these tasks step by step, incrementally. Along the chapters, we will setup a project that you can use as a template later.

We won't be covering *everything* you can do with Grunt, but you should be able to figure it out by yourself once you finish all the chapters. The goal of this guide is to give you a **solid basis** so you can look out how to automatise what you need *exactly*.

Ah! And Grunt is free and open source. The code is available in a repository on Github[4]. The community is growing and there are hundreds of plugins you can use.

Install Node and Grunt

You might not have yet Node.js[5] installed in your system. **Node** is a system that can run JavaScript *outside* a browser, like a regular scripting language.

You can install Node both from source code or binaries. Just go to the downloads page[6] and grab a distribution appropriate for your system. In addition to those downloads, if you are on a Mac you can install Node with Homebrew[7]:

[3]http://www.gruntjs.com
[4]https://github.com/gruntjs/grunt
[5]http://www.nodejs.org
[6]http://nodejs.org/download/
[7]http://brew.sh/

```
$ brew install node
```

After installing Node, you need to install NPM, a package manager for Node code (think of it as Node's RubyGems). You can install NPM from source:

```
$ curl http://npmjs.org/install.sh | sh
```

Are you using Windows?

In this book we are assuming you have **access to a shell**. A *real* shell. Taking into account all the tools that UNIX systems provide to developers, please consider installing a Linux distribution in a virtual machine.

If this sounds too much, **you can still install Node and Grunt in Windows**… But keep in mind that all the commands shown to create directories, print files' contents, etc. work only in UNIX systems. You should use Windows-equivalent commands when following the examples.

Now that we have both Node and NPM, we can install Grunt. There are two packages that we need to download. One of them, `grunt-cli` contains Grunt's command line interface, so you can run grunt in your console. The other one, `grunt`, is the actual code and you need to install it per project (as we'll do later).

`grunt-cli` needs to be **installed globally**, like this:

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

Start the project

We mentioned a **project** we would be working on along the chapters in this book. We are starting it now! Open a terminal and create a `grunt-tutorial` directory wherever you want:

```
$ mkdir grunt-tutorial
$ cd grunt-tutorial
```

NPM has a cool way of managing dependencies and project's metadata. You just need to create a file named `package.json` in the root of your project and let the magic happen.

The initial contents of this JSON file are as follows:

```
{
  "name": "grunt-tutorial",
  "version": "0.0.0"
}
```

Now we will install grunt in this project, via NPM.

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

See that `--save-dev` flag? This tells NPM to update the `package.json` file to add the `grunt` module as a **dependency**. Take a look at its contents now:

```
$ cat package.json
{
  "name": "grunt-tutorial",
  "version": "0.0.0",
  "devDependencies": {
    "grunt": "~0.4.1"
  }
}
```

This also creates a `node_modules` directory: all the Node packages that you install will be copied into this directory. The dependencies list is very useful (it works like a Gemfile), specially when we are working with more people in a project.

If you run `npm install`, NPM will read the dependencies list and will **install everything** that is listed, with the appropriate version. Let's do a test by deleting are newly-installed grunt, and re-installing it again:

```
$ rm -rf node_modules/*
$ npm install
$ tree -L 2
.
├── node_modules
│   └── grunt
└── package.json
```

Last, we need to create a special file in the root of our project, called `Gruntfile.js`. This is the file that Grunt will load and will get the tasks code from.

For now, we will leave this file empty:

```
$ touch Gruntfile.js
```

If everything has gone well, you should be able to **run Grunt** and see an error of not having a default task to run:

```
$ grunt
Warning: Task "default" not found. Use --force to continue.
Aborted due to warnings.
```

Don't worry, we'll fix this lack of tasks in the next chapter.

Recap

In this chapter we have learnt:

- What is Grunt and why we want to use it.
- How to install Node and NPM.
- How to install Grunt.
- How to start a project that uses Grunt from scratch.

The code so far

You can download the full source code for this chapter at the book's home page[8].

package.json (after installing Grunt)

```
{
  "name": "grunt-tutorial",
  "version": "0.0.0",
  "devDependencies": {
    "grunt": "~0.4.1"
  }
}
```

[8]http://www.belenalbeza.com/books/grunt

Add a linter

A **linter** is a tool that analyses your code and spots not just syntax errors, but dangerous code practises and bad formatting.

Since JavaScript is a scripted language, running a linter will save us lots of time because it will inform us about **syntax errors and dangerous code**. Aren't you tired of opening a browser to find out you forgot a brace or a parenthesis? Or even worse, that you mistyped a variable name, causing a bug in your code, since JavaScript allows creating variables on the fly?

A linter will warn you about these situations, and even more: it also controls **formatting**, so the code is consistent about the use of whitespace, indentation, and line length. This is a must-have in your project: it will save you time, and prevent bugs.

Install the JSHint plugin for Grunt

The linter we will use is JSHint[9]. And there's already a Grunt **plugin** cooked for us: `grunt-contrib-jshint`. Let's start by installing it:

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

Now we need to think about which files do we want to run through JSHint. This is, obviously, our JavaScript files, but where do we put them?

We have not spoken yet about the **file structure** that our web app will have… This is something we will be building along this guide, step by step.

For the moment, we will have an `app` directory in our root folder: here we will put all of our app assets and files. Why create a separate directory for this? Because ideally we will have more stuff hanging from root that it is not strictly our web app itself: a `README` file, rules for the linter, a `test` folder, etc.

Inside this `app` directory we will have a `js` directory, where we will write our JavaScript files. And inside this `js`, we will have another directory, called `vendor`, where we will drop third-party libraries or legacy code we have inherit.

[9]http://www.jshint.com

```
$ mkdir app
$ mkdir app/js
$ mkdir app/vendor
```

Now our project directory should look like this (ignoring the `node_modules` folder):

```
$ tree -I node_modules
.
├── Gruntfile.js
├── app
|   └── js
|       └── vendor
└── package.json

3 directories, 2 files
```

Lint a single file

Now we are going to edit the `Gruntfile` to **load the linter task**:

```
'use strict';

module.exports = function (grunt) {
  // load jshint plugin
  grunt.loadNpmTasks('grunt-contrib-jshint');
};
```

This is very basic `Gruntfile`. Here we are just loading the `grunt-contrib-jshint` plugin, that contain a `jshint` task already written for us. Let's try to run it:

```
$ grunt jshint
>> No "jshint" targets found.
Warning: Task "jshint" failed. Use --force to continue.

Aborted due to warnings.
```

Here Grunt is complaining about no having any targets for the task named `jshint`. **What's a target**? It's a core concept of Grunt. When we create a task, we add targets to it. Every target represents a set of actions and files the task will be run over. We can run a task's target by simply appending it to the task name.

```
$ grunt mytask:mytarget
```

If we omit the target, *all targets* associated with that task will be run.

For now, we will just add a single target for our task, and put an arbitrary name: `all`. The way we add targets and other **configuration options** is by using the `grunt.initConfig` method and passing a configuration `Object` to it:

```
'use strict';

module.exports = function (grunt) {
  // load jshint plugin
  grunt.loadNpmTasks('grunt-contrib-jshint');

  grunt.initConfig({
    jshint: {
      all: [
        'Gruntfile.js'
      ]
    }
  });
};
```

There are several ways to setup our tasks' targets –here is one of them: we are passing an array of file names that our task we will use. In our case, JSHint will scan these files.

If we try it now, we will scan our `Gruntfile`... and get some errors:

```
$ grunt jshint
Running "jshint:all" (jshint) task
Linting Gruntfile.js...ERROR
[L1:C1] W097: Use the function form of "use strict".
'use strict';
Linting Gruntfile.js...ERROR
[L3:C1] W117: 'module' is not defined.
module.exports = function (grunt) {

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

Both errors are because the `Gruntfile` is a Node program, and by default JSHint does not recognise or allow the use of `module` and the string version of `use strict`. We can set a JSHint **rule** that will accept our Node programs. Let's edit our `jshint` task configuration and add an `options` key:

```
jshint: {
  options: {
    node: true
  },
  all: [
    'Gruntfile.js'
  ]
}
```

Even though `options` looks like another target, it is not one. This is a reserved name in Grunt and is the way to setup some configuration options that will be either shared among all the targets (when they are at the top level), or be specific to a target (when they are under that target).

Let's try this now:

```
$ grunt jshint
Running "jshint:all" (jshint) task
>> 1 file lint free.

Done, without errors.
```

Hooray! It works!

Lint the whole project

Right now we are only linting the `Gruntfile`, but we want to lint all of our JavaScript. For that, we just need to add to the task's target all the files that are inside the `app/js` directory.

We don't want to be listing all the files one by one. Luckily, Grunt uses patterns to match files in tasks. Grunt uses the `minimatch` library to match these patterns against the file names, so check out the documentation[10] for more info.

For instance, `'app/js/*.js'` will match all the files with `.js` extension that are included on `app/js`, but *only at the root level*. If we want to include subdirectories as well, we need to use a double asterisk to match any subdirectories (including sub-sub-sub-… directories), like this: `'app/js/**/*.js'`.

Edit the `Gruntfile` to add this change:

[10] https://github.com/gruntjs/grunt/wiki/Configuring-tasks#files

```
jshint: {
  options: {
    node: true
  },
  all: [
    'Gruntfile.js',
    'app/js/**/*.js'
  ]
}
```

Now drop a file in `app/js` to actually try it:

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

Done, without errors.
```

Edit the file you just added and type some incorrect JavaScript. For instance:

```
console.log('hola); // here we are forgetting a final '
```

If you run `jshint` again, you will see your error pointed out:

```
grunt jshint
Running "jshint:all" (jshint) task
Linting app/js/main.js...ERROR
[L1:C13] W116: Expected ')' and instead saw ''.
console.log('hola); // here we are forgetting a final '
Linting app/js/main.js...ERROR
>> Missing semicolon.

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

So this seems to be working! But there's still something not quite well. Remember the `vendor` directory? If we put here **third-party code** we have not written ourselves, we don't want to run the linter on it. Fortunately, this is easily solved by using an *ignore pattern*: it's like a regular pattern but preceded with a `!`:

```
jshint: {
  options: {
    node: true
  },
  all: [
    'Gruntfile.js',
    'app/js/**/*.js',
    '!app/js/vendor/**/*.js'
  ]
}
```

Now move our file with errors from `app/js` to `app/js/vendor`, and run `jshint`:

```
$ mv app/js/*.js app/js/vendor/
$ grunt jshint
Running "jshint:all" (jshint) task
>> 1 file lint free.

Done, without errors.
```

See? No errors this time, and only one file has been scanned. This is because JSHint is effectively ignoring the files inside `vendor`, like we told it.

Customise the linter's rules

JSHint comes with a **preset of rules** that will apply when scanning the files. We can add or remove rules to this default preset to adapt the linter to our needs. We actually did this before: by enabling Node code in the `options` key when configuring the `jshint` task.

But there's a more clean way to do this: by putting all the rules in a JSON file and tell the `jshint` task to load this file and use it as rules. By convention, this file is usually named `.jshintrc`. So let's create it in the root of our project:

```
{
  "node": true
}
```

After this, we just need to tell `jshint` to use this file. We do this in the `options` key:

```
jshint: {
  options: {
    jshintrc: '.jshintrc'
  },
  all: [
    'Gruntfile.js',
    'app/js/**/*.js',
    '!app/js/vendor/**/*.js'
  ]
}
```

Now we can try it:

```
$ grunt jshint
Running "jshint:all" (jshint) task
>> 1 file lint free.

Done, without errors.
```

It's time to pick which rules we want to include or remove. You can see a full list at [the JSHint documentation](http://jshint.com/docs/options/, but here's the set of rules I like to use:

```
{
  "bitwise": true,
  "browser": true,
  "camelcase": true,
  "curly": true,
  "esnext": true,
  "eqeqeq": true,
  "eqnull": true,
  "immed": true,
  "indent": 2,
  "latedef": true,
  "maxlen": 80,
  "maxstatements": 20,
  "newcap": true,
  "node": true,
  "strict": true,
  "trailing": true,
  "quotmark": "single",
  "undef": true,
  "unused": true,
  "white": true,
  "predef": [
    "alert"
  ]
}
```

The rules above are pretty strict and won't allow us to create a variable and not using them, and it also checks whitespace formatting. In addition to that, there's a limit in the number of characters per line, as well as in the number of statements per method (this is good to keep code complexity at bay).

There's an important rule, called `predef`. There we have an array of identifiers we want to enable globally. By adding `alert` we can use the `alert` method instead of having to write `window.alert`. If you're using Zepto or jQuery you might want to add `$` to this array (the same for _ if you're using Underscore). Of course, the proper way would be to use AMD modules (pretty easy with RequireJS), but…

Recap

In this chapter we have learnt:

- What is a linter and why it's useful.
- How to install a Grunt plugin.
- How to configurate a Grunt task with targets.
- How to run a Grunt task.
- How to customise JSHint's ruleset.

The code so far

You can download the full source code for this chapter at the book's home page[11].

.jshintrc

```
{
  "node": true
}
```

[11]http://www.belenalbeza.com/books/grunt

Gruntfile.js

```
'use strict';

module.exports = function (grunt) {
  // load jshint plugin
  grunt.loadNpmTasks('grunt-contrib-jshint');

  grunt.initConfig({
    jshint: {
      options: {
        jshintrc: '.jshintrc'
      },
      all: [
        'Gruntfile.js',
        'app/js/**/*.js',
        '!app/js/vendor/**/*.js'
      ]
    }
  });
};
```