Front-End FUNDAMENTALS

Joe Fender | Carwin Young

Front-End Fundamentals

A practical guide to front-end web development.

Joe Fender and Carwin Young

This book is for sale at http://leanpub.com/front-end-fundamentals

This version was published on 2015-05-06

Tweet This Book!

Please help Joe Fender and Carwin Young by spreading the word about this book on Twitter!

The suggested hashtag for this book is #front-end-fundamentals.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#front-end-fundamentals

Thanks to,

Sally Rose, Carrie Young and Justin Harrell

Contents

About the Authors

Joe and Carwin started working together in 2013 at Lullabot[1], an interactive strategy, design and development company.

At Lullabot, they work with decision makers, developers, and designers at all stages of a web project. They have guided the development of sites for clients such as Martha Stewart, The GRAMMYs, MTV UK, Sony Music, Lifetime Television, The George Lucas Educational Foundation, Leo Laporte's TWiT network, and many others.

Joe Fender

Joe Fender lives in London, UK and splits his focus between front-end and back-end development. Specializing in PHP, AngularJS and Drupal, Joe was the lead architect at Studio Umi[2], the largest Drupal shop in Japan before joining Lullabot to work on awesome websites such as Drupalize.Me[3] and MSNBC[4].

In his spare time, Joe likes to collect shoes and work on side projects. But never at the same time.

[1]https://www.lullabot.com/
[2]http://www.studio-umi.jp/english
[3]http://www.drupalize.me
[4]http://www.msnbc.com

Carwin Young

Carwin Young is an expert front-end developer living in Springfield, Missouri. He has over 8 years experience working on the front-end. He was the second front-end developer to work at Lullabot where he has played a key role on building websites for clients such as MSNBC, BravoTV and the GRAMMYs.

Carwin loves to share his expertise. He has spoken on topics such as Rapid Drupal Development at DrupalCamp Chicago, is a co-organizer for the Springfield Drupal meet-up and is a regular contributor to the Front-end Rapport[5], an online magazine for front-end developers.

[5]https://flipboard.com/section/front-end-rapport-bML9IT

Introduction

Front-end web development is evolving. Long gone are the days of writing static HTML files with self-contained CSS. As web developers in this modern world, we need to understand and utilize an endless number of frameworks, plugins, techniques and more. Knowing the right tools to use and when to use them is key to building successful solutions.

Despite our best efforts, it can be difficult to keep up with the speed of advancing web development. We sure as hell can't slow it down. And why would we want to? It's so exciting to not know where technology is headed and what could be around the next corner. We should both prepare for change and embrace it. And that is exactly where front-end developers excel. We build what the consumer sees and because of that, we want to use the most cutting-edge technologies to create the biggest impact.

This book came to be from the lessons we, the authors, have learnt through trial and error whilst working on front-end development projects ranging from enormous high-traffic sites such as MSNBC, The GRAMMYs and BravoTV to startups and experiments such as Drupalize.Me and BracketCloud. We want to share with you the tools that we use and the standards that we follow. Our ultimate goal is for you to walk away understanding the core concepts of front-end development so you can confidently go and work on your own projects.

This book is somewhat opinionated based on our experience, so you'll find some of the content heavily geared towards a particular way of doing things. Of course, there are many ways of accomplishing the same task in the world of front-end development; that's what makes it so accessible and yet so complicated. Learning one way of doing something can make learning the alternatives a much more palatable task, so we encourage you to look into any and all of the technologies presented herein and even go on to see if something else might better fit your needs.

Who This Book Is For

The content in this book is aimed at those who are new to front-end web development. Regardless of whether you are a beginner or expert developer, the concepts outlined in this book are essential to embracing the rapidly evolving web. We won't be covering the general basics of web development so you will most likely want to get up to speed on the fundamentals of web programming such as HTML, CSS and JavaScript. We will also be utilizing the command line, so some basic knowledge of how to use that on your operating system is recommended.

If you hear yourself asking any of the following questions, this book will likely be very useful for you!

- "What actually is front-end development?"

- "How can I be a more efficient web developer?"
- "What is Grunt/AngularJS/SASS/<*insert popular tool name here*>?"
- "How do I fit all of these front-end tools together into a project?"
- "How can I build a front-end application that can scale?"

An Overview of This Book

Every web project that you work on is different and there is no 'one size fits all' set of tools. However, throughout the chapters in this book, we'll introduce you to some of the most popular, useful and powerful tools used in front-end web development so that when you're done reading you will be able to apply the concepts and techniques to whatever you are working on.

1. Getting Started

The aim of the first chapter of this book is to help get you acclimated with front-end development in general. You'll learn what's expected from a front-end developer and get a short overview of the types of projects a front-end developer might be tasked with. We'll even take you through setting up your computer for local development.

2. Frameworks

This chapter jumps right into some of the more popular JavaScript frameworks and template languages in use today. We'll cover the details of what exactly a 'framework' is and talk about the MVC pattern before getting into the details of installation and practical usage of 3 popular frameworks: AngularJS, Backbone.js and Ember.js.

3. Styling

There's a lot more to styling than just throwing CSS at your markup; that just leads to a big mess of unreadable spaghetti code sprinkled with `!important` parmesan. Here we'll cover CSS preprocessors like Sass, organizational methods for your styles, and the kind of impact your CSS may have on a project's performance.

4. Dependency Management

Getting your whole team on the same page code-wise can be challenging. Heck, even making sure you're using the same version of that awesome Ruby gem after an update can be a pain. This chapter on dependency management covers a number of really handy front-end package managers like `npm`, `Bower`, and `Bundler`. You'll learn what a package manager is and how to use it effectively to keep your project on track.

5. Automation

Do you like doing the same repetitive tasks over and over? No? Neither do we.

The automation chapter is about just that, automation. We'll go over how you can remove the tedium from front-end development and just get to work. In this chapter you'll learn how to perform tasks like checking your JavaScript or CSS for errors whenever you save a file, minify your code to shave off some file size, and even how to automatically refresh your browser whenever something changes. Once you get started, you'll never look back.

Example Code

This book contains many code snippets for demonstration purposes. Code may appear differently depending on the device you are using to read this book. Here is an example of a code snippet so you know what to expect:

```
var gulp = require('gulp'),
  uglify = require('gulp-uglify');

// This task uglifies our JS files.
gulp.task('compress', function() {
  gulp.src('src/js/*.js')
    .pipe(uglify())
    .pipe(gulp.dest('dist'))
});
```

You will also notice that we reference code inline `like this` throughout the book.

Contacting the Authors

If you have any feedback or questions, please feel free to contact us via the Front-End Fundamentals Google Group[6].

You can either create a new topic via the web interface or you can send an email to the mailing list address: front-end-fundamentals@googlegroups.com[7]

We will do our best to consider each and every new message received. Please understand that depending on the volume of requests, it may be difficult for us to respond immediately.

[6]https://groups.google.com/forum/#!forum/front-end-fundamentals
[7]mailto:front-end-fundamentals@googlegroups.com

SAMPLE

Automation

Time is a key factor in productivity. This chapter will explore some ways to automate your development workflow so you spend less time working and more time making awesome. You'll be introduced to 'task runners' which help automate repetitive tasks. You'll also discover ways to speed up the process of laying foundations for new projects, and you'll pick up a few other tips to help you be a more efficient developer.

Task Runners

Task runners, also known as 'task automation frameworks', help you manage and execute sets of defined tasks. A task can be seen as any repetitive action you have to take when developing for the web. Tasks can include actions such as copying a file between directories or checking your JavaScript for syntax errors.

With a task runner, you can build, preview and test your application much more easily than if you did all the work yourself. The strongest attribute of task runners is their flexibility, which means you can configure and run pre-defined tasks exactly how you want. Furthermore, if you can't find a pre-defined task that meets your needs, you can even write your own.

In this chapter you'll learn a bit about the popular task runners Grunt, Gulp, CodeKit and Yeoman, including how to install them and how to configure them to meet your specific needs.

Grunt

Grunt is one of a front-end developer's best friends and is the most popular and widely used task runner. It is a framework based on JavaScript and has a thriving ecosystem. In addition, there are a huge number of plugins available for Grunt. If you're not sure where to start with task runners, then start here.

Installing Grunt

Let's work through the installation together. We'll install Grunt using `npm` from the command line. (If you haven't done so already, please read the *Dependency Management* chapter first to learn more about how to set up and use `npm`.)

The first tool we're going to install is Grunt's command line interface, `grunt-cli`. It's worth noting that this does not actually install Grunt itself. It only provides us global access to the `grunt` keyword from the command line which is then used to load and run the version of Grunt that has been installed locally within the project. It is important to remember to install `grunt-cli` globally but `grunt` locally. For more information on global vs local installation, check out this article[8] over at the official NodeJS blog.

We'll head over to our command line and run:

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

To verify that Grunt CLI was installed successfully and that we can access the `grunt` command we'll run the following:

```
grunt --version
```

Now let's install Grunt. We'll need to navigate to our project's root directory first:

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

This will install Grunt locally and save a dependency reference to it from within the `package.json` file.

The Gruntfile

Now that we've installed Grunt, it's time to introduce `Gruntfile.js`. Like `package.json` for `npm`, the Gruntfile belongs in the root directory of our project. It defines our project and task configurations and tells Grunt which plugins we want to use to run these tasks. We're not going to define any tasks just yet but let's create the bare bones of the Gruntfile so that we can add to it later. Create `Gruntfile.js` in the project's root directory and add the following:

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

  // Default tasks.
  grunt.registerTask('default', []);
};
```

[8]http://blog.nodejs.org/2011/03/23/npm-1-0-global-vs-local-installation

You'll notice that this file, as the file name suggests, is a JavaScript file. All we've done here is called `initConfig` which initializes our Grunt configuration. This is where we will define our tasks later. We've also called `registerTask` to initialize a set of tasks labelled `default`. There are no actual tasks here yet but Grunt requires us to define this for it to run. After saving `Gruntfile.js` we can try running `grunt` in the command line from the project root. We should get a shiny green message stating:

```
Done, without errors.
```

This means Grunt was able to run all of the tasks defined in default without failure.

Yay! We've just run Grunt for the first time. The more you use Grunt, the more you'll love it!

Popular Plugins

There are thousands of plugins available in the open source community for Grunt. But there are a few that have become staples for the modern front-end developer. Depending on the project, you may not need all of these, but it doesn't hurt to know about them. Obviously, these are only a select few and may not include everyone's favorites.

Watch, LiveReload and Connect

Quite possibly the most popular plugin is Watch. It allows you to run specified tasks whenever a watched file changes. So, for example, you could minimize your CSS files whenever one of them changes or run JavaScript tests whenever you save script changes. The Watch plugin only does the 'watching' part. You have to specify exactly which tasks are to be run when a file change is detected.

To install Watch, we'll run the following from the command line:

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

Edit the `Gruntfile.js` so that the plugin gets loaded:

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

  // Load our Grunt plugins.
  grunt.loadNpmTasks('grunt-contrib-watch');

  // Default tasks.
  grunt.registerTask('default', []);
};
```

All we have done here is use the `loadNpmTasks` callback to tell Grunt to load the Watch plugin.

To demonstrate how to use Watch, let's set up something cool using LiveReload and Connect. We'll set up a local web server that will automatically refresh any browser viewing our web site when a file change occurs. This is great because it means we don't have to refresh every time a change is made to one of the files. The official homepage for LiveReload describes the tool as "A happy land where browsers don't need a refresh button"!.

Usually, when setting up a local web server you have to configure software system-wide. You may have used tools such as MAMP or you may have configured a local LAMP stack yourself. But a good alternative is to use a more lightweight approach to testing web applications using a combination of Watch, Connect and LiveReload plugins.

Connect is an HTTP web server framework built for NodeJS. It's a lightweight static web server that provides plugins known as 'middleware'. Connect comes with a bunch of middleware such as session support, a cookie parser and more.

There is a Grunt plugin for Connect which allows us to start a Connect web server so let's get that installed:

```
npm install grunt-contrib-connect —save-dev
```

We now need to tell Grunt when and how to use the plugin. Every plugin you install should have a README file available in its root directory (in this case, `node_modules/grunt-contrib-connect/README.md`) which will outline how to configure the task. Let's edit our `Gruntfile.js` again to set up the `connect` task:

```
module.exports = function(grunt) {
  // Project configuration.
  grunt.initConfig({
        // Defines our Connect web server.
    connect: {
      server: {
        options: {
          port: 9000,
          hostname: 'localhost',
          base: ['dist/']
        },
      },
    },
  });

  // Load our Grunt plugins.
  grunt.loadNpmTasks('grunt-contrib-watch');
  grunt.loadNpmTasks('grunt-contrib-connect');

  // Default tasks.
  grunt.registerTask('default', []);

  // Running `grunt server` runs our Connect server until terminated.
  grunt.registerTask('server', ['connect:server:keepalive']);
};
```

Here, we told Grunt to load the plugin using `loadNpmTasks` exactly as we did with Watch. Task configurations are added within the `initConfig` callback. We've defined a task called `connect` that has a sub-task called `server` containing the settings for the task. We've defined our Connect server to run on port 9000 at `http://localhost:9000` and to serve the folder `dist` (short for distribution).

As described in the code comment near the bottom, running `grunt server` from the command line will launch the Connect server until terminated (made possible through the keepalive parameter). You will then be able to access the web application by pointing your browser to `http://localhost:9000`. However, there is currently no dist folder so let's move on to creating that and an example `index.html`:

```
mkdir dist
cd dist
touch index.html
```

Open up `index.html` to add some basic HTML such as the following:

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>My App</title>
  </head>
  <body>
    <p>Hello world! Welcome to my application.</p>
  </body>
</html>
```

Now, if you go to the command line, run `grunt server` and then point your browser to `http://localhost:9000` you should see the "Hello world" message.

To finish, we need to configure the Watch task so that when any change occurs in `index.html`, a browser refresh is triggered. Let's update our `Gruntfile.js` to look something like this:

```
// Project configuration.
grunt.initConfig({
  // Defines our Connect web server.
  connect: {
    server: {
      options: {
        port: 9000,
        hostname: 'localhost',
        base: ['dist/'],
        livereload: true
      },
    },
  },

  // Watch our files for changes.
  watch: {
    options: {
      livereload: true
    },
    html: {
      files: ['dist/index.html'],
      tasks: []
    }
  },
});

// Load our Grunt plugins.
grunt.loadNpmTasks('grunt-contrib-watch');
grunt.loadNpmTasks('grunt-contrib-connect');

// Default tasks.
grunt.registerTask('default', ['watch']);

// Running `grunt server` runs our Connect server until terminated.
grunt.registerTask('server', ['connect:server:keepalive']);
```

The first thing you may notice is we've added our `watch` task configuration. We've set up a sub-task called `html` that watches for changes to `index.html`. We haven't actually told Watch to execute any specific tasks once a change is detected. However, we've enabled the `livereload` boolean both in the Connect and Watch options. This basically tells Connect and Watch to use LiveReload in tandem. Connect comes complete with the connect-livereload middleware Node package and, as of recently, Watch now comes with LiveReload server functionality included so all we need to do is set those two booleans. Lastly, we updated our `default` task so that the `watch` task is run.

To test this, you're going to need two terminal windows. In the first, run `grunt server` and in the second run `grunt`. You should see the following status messages respectively:

```
Running "connect:server:keepalive" (connect) task
Waiting forever...
Started connect web server on http://localhost:9000
```

And:

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

Make sure your browser is looking at `http://localhost:9000` and then make a change to the "Hello world" message in your `index.html`. Don't forget to save the file! Once you save the file, switch back over to your browser and you'll see that the page had refreshed (probably before you even made it back over) and reflects the change you made. Neat huh?

The power of Watch really shines when you start to configure it for larger front-end applications. For many, it becomes the center of their automation workflow - the trigger to every task. It really is a great plug-in.

Copy and Concat

As programmers, we aim to develop our projects to be as modular as possible. This quickly leads to a lot of files and one of the biggest challenges is keeping on top of them all. Copy and Concat are a couple of the handiest file management plugins to know about.

Copy

Copy simply allows you to copy files and folders from Grunt. Copy has endless uses but for the purposes of demonstration let's create an example scenario. Let's say that we want to copy `src/index.html` to `dist/index.html` as part of our Grunt build process because the `dist` folder is the one that is to be served by our web server and `src` is where we actually edit our code. Obviously with a real application there would likely be more files to copy than just a single file but let's roll with this anyway.

Before getting started, don't forget to install the plugin:

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

This is how we might go about setting up our `Gruntfile.js` to reflect this scenario:

```
module.exports = function(grunt) {
  // Project configuration.
  grunt.initConfig({
    copy: {
      html: {
        files: [
          {
            src: ['src/index.html'],
            dest: 'dist/index.html'
          }
        ]
      }
    },
  });

  // Load our Grunt plugins.
  grunt.loadNpmTasks('grunt-contrib-copy');

  // Default tasks.
  grunt.registerTask('default', ['copy:html']);
};
```

We've set up a sub-task within our `copy` task configuration called `html` which will copy `index.html` for us. This allows us to add the `copy:html` task to our `default` task list. Simply entering `copy` as the task name would have also worked as that would trigger Grunt to execute all of its sub-tasks, but it's preferable to explicitly call the name of the sub-task you wish to execute to prevent any issues in the future when you expand on your configurations.

To test, make sure that the `src/index.html` file exists and then simply run `grunt` from the command line. You should see the following success message:

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

Done, without errors.
```

To specify more advanced file paths, such as using wildcards and matching on file extensions, read the official documentation on Globbing Patterns[9] within Grunt. Copy also has some useful options such as `encoding` to use a specific file encoding when copying or `mode` to specify copied file permissions.

Concat

[9]http://gruntjs.com/configuring-tasks#globbing-patterns

Concat lets you concatenate files. Concatenation is the act of joining things together. When optimizing your applications for performance, looking at techniques to reduce the number of requests is extremely important. By concatenating files together, you are able to reduce the number of requests required by clients to get at your code.

To install Concat, we'll run the following from our project folder:

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

Let's say we want to concatenate all of the JavaScript files that we have within the folder `src/js` into a single file. This is how we would go about setting up our `Gruntfile.js`:

```
module.exports = function(grunt) {
  // Project configuration.
  grunt.initConfig({
          // Concatenate all JavaScript files.
    concat: {
      js: {
        src: ['src/js/*.js'],
        dest: 'dist/scripts.js'
      },
    },
  });

  // Load our Grunt plugins.
  grunt.loadNpmTasks('grunt-contrib-concat');

  // Default tasks.
  grunt.registerTask('default', ['concat:js']);
};
```

You can test this by adding a bunch of JavaScript files in to the `src/js` folder and then running `grunt`. You may notice that Concat could possibly nullify the need for Copy as you can specify the destination of the concatenated file. However, if you wanted to run other tasks on the concatenated file before moving it to the `dist` folder, you may well still need Copy.

One configuration option of Concat that you can use is `banner`. It allows you to set a string header that will be prepended to the beginning of the concatenated file. It is worth noting that the header string is processed using grunt.template.process[10].

Here is an example of how to set up a banner in `Gruntfile.js` for a typical application:

[10]https://github.com/gruntjs/grunt-docs/blob/master/grunt.template.md#grunttemplateprocess

```
module.exports = function(grunt) {
  // Project configuration.
  grunt.initConfig({
    pkg: grunt.file.readJSON('package.json'),

    // The banner is placed at the head of our concatenated JavaScript file.
    banner: '/*! \n* <%= pkg.title || pkg.name %> - v<%= pkg.version %>' +
            '\n* Copyright (c) <%= grunt.template.today("yyyy") %> <%= pkg.autho\
r %> ' +
            '\n* <%= pkg.homepage ? pkg.homepage : "" %> ' +
            '\n*/ \n\n',

    // Concatenate all JavaScript files.
    concat: {
      options: {
        banner: '<%= banner %>'
      },
      js: {
        src: ['src/js/*.js'],
        dest: 'dist/scripts.js'
      },
    },
  });

  // Load our Grunt plugins.
  grunt.loadNpmTasks('grunt-contrib-concat');

  // Default tasks.
  grunt.registerTask('default', ['concat:js']);
};
```

What this does is load in the `package.json` JSON file into a variable called `pkg` which is then used to fill out `banner`. Depending on how your `package.json` is set up, you should see something like this at the top of the outputted `scripts.js` file:

```
/*!
* myApp - v0.0.0
* Copyright (c) 2014 Joe Fender
* http://www.myapp.com
*/
```

Linting with JSHint and CSSLint

Linting is the process of analyzing code for potential errors. In traditional methods, you would see JavaScript errors in the console if something went wrong. However, by linting your code, you are able to get feedback on your coding quality directly within your Grunt build process. Most linters will give you more information about syntax errors than your traditional browser console. Linting your code is highly recommended as it can save a lot of time debugging syntax errors.

JSHint

For JavaScript, we use grunt-contrib-jshint[11]. To install JSHint, we'll run this from our project folder:

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

This Grunt plugin passes through all of the options you set in its configuration task directly to JSHint. That means you will want to be reading the official JSHint documentation[12] for a list of supported options.

Here is an example of how we go about setting up our own typical JSHint configuration in `Gruntfile.js`:

```
module.exports = function(grunt) {
  // Project configuration.
  grunt.initConfig({
    pkg: grunt.file.readJSON('package.json'),

    // Configures JavaScript linting. See http://www.jshint.com/docs/options/
    // for more on the available options.
    jshint: {
      js: {
        files: {
          src: ['src/js/*.js']
        },
        options: {
          curly: true,
          immed: true,
          newcap: true,
          noarg: true,
          sub: true,
          boss: true,
          eqnull: true,
          strict: false,
```

[11]https://www.npmjs.org/package/grunt-contrib-jshint
[12]http://www.jshint.com/docs/options/

```
        globalstrict: true,
        globals: {
          angular: false
        }
      }
    }
  }
  });

  // Load our Grunt plugins.
  grunt.loadNpmTasks('grunt-contrib-jshint');

  // Default tasks.
  grunt.registerTask('default', ['jshint']);
};
```

We've told JSHint to check all of the JavaScript files in the `src/js` folder as part of our `default` task list. To test this, create an example JavaScript file that has a deliberate syntax error. Such as:

```
var example = {
  bar: 'bar',
  foo: 'foo'
}
```

Run `grunt` and you will see the following:

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

   src/js/example.js
      4 |}
          ^ Missing semicolon.

>> 1 error in 1 file
Warning: Task "jshint:files" failed. Use --force to continue.

Aborted due to warnings.
```

JSHint found the missing semicolon and gave us a very clear indication of where the syntax error was. Because there was an error, it aborted the Grunt build process.

If you have a build process that takes several seconds to complete, bring your linter task as far towards the front of the list as you can so that you'll get snappier feedback about possible syntax errors. That way, you're not wasting Grunt's or your own time going through a bunch of tasks that you'll just have to go through again once you fix the syntax error.

CSSLint

For CSS, grunt-contrib-csslint[13] is recommended. As with JSHint, view the CSSLint documentation[14] for a full list of configuration options. To install CSSLint we'll run this command:

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

Here is an example `Gruntfile.js` to get CSSLint working with default settings:

```
module.exports = function(grunt) {
  // Project configuration.
  grunt.initConfig({
    // Configures CSS linting. See https://github.com/CSSLint/csslint
    // for more on the available options.
    csslint: {
      css: {
        src: ['src/css/*.css']
      }
    }
  });

  // Load our Grunt plugins.
  grunt.loadNpmTasks('grunt-contrib-csslint');

  // Default tasks.
  grunt.registerTask('default', ['csslint']);
};
```

To test CSSLint create a broken css file in `src/css` - perhaps something like this:

```
.boo {
  height: 100px;
  widths: 100px;
}
```

After running `grunt` you should be greeted with this lovely warning message:

[13] https://www.npmjs.org/package/grunt-contrib-csslint
[14] https://github.com/CSSLint/csslint

```
Running "csslint:css" (csslint) task
Linting src/css/example.css...ERROR
[L3:C3]
WARNING: Unknown property 'widths'. Properties should be known (listed in CSS3
specification) or be a vendor-prefixed property. (known-properties)
Browsers: All
>> 1 file lint free.

Done, without errors.
```

It is worth noting that unlike JSLint, the Grunt build process will not fail when there is a code warning. This seems to be a design decision.

Minification with UglifyJS and CSSMin

Minification is the process of removing unnecessary characters from code (such as white space and comments) to reduce its file size. There are lots of plugins available that offer different types of minification but a couple of favorites are UglifyJS and CSSMin.

UglifyJS

UglifyJS[15] is a JavaScript compressor and minifier. To install UglifyJS, we'll run this command from our project root:

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

We can set up the `uglify` task in our `Gruntfile.js` with only a few lines:

```
module.exports = function(grunt) {
  // Project configuration.
  grunt.initConfig({
    // Minifies our JavaScript files.
    uglify: {
      js: {
        files: {
          'dist/scripts.min.js': ['src/js/*.js']
        }
      }
    },
  });

```

[15]http://lisperator.net/uglifyjs/

```
  // Load our Grunt plugins.
  grunt.loadNpmTasks('grunt-contrib-uglify');

  // Default tasks.
  grunt.registerTask('default', ['uglify']);
};
```

In the above example we've configured our `uglify` task to look for all JavaScript files in `src/js` and compress them all into the outputted `dist/scripts.min.js`. By default, UglifyJS will not only remove white space and comments but will also try to simplify the code by grouping together code of similar functionality (such as variable declarations) and by deleting unused functions.

Check out all of the task configuration options available in the plugin README, especially `mangle`. By setting this to `true`, UglifyJS reduces the names of local variables and functions to single letters. This can greatly minimize the amount of text used in the JavaScript file, thus reducing the outputted file size.

CSSMin

CSSMin[16] is probably the most popular CSS minifier Grunt plugin. It uses clean-css[17] to perform the actual compression.

To install CSSMin, we'll run this command from our project root:

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

Here is an example `cssmin` task for our `Gruntfile.js` that will take all of the CSS files in `src/css` and minify them into a single file, `dist/styles.min.css`:

```
module.exports = function(grunt) {
  // Project configuration.
  grunt.initConfig({
    // Minifies our CSS files.
    cssmin: {
      css: {
        files: {
          'dist/styles.min.css': ['src/css/*.css']
        }
      }
    },
  });

```

[16]https://github.com/gruntjs/grunt-contrib-cssmin
[17]https://github.com/jakubpawlowicz/clean-css

```
  // Load our Grunt plugins.
  grunt.loadNpmTasks('grunt-contrib-cssmin');

  // Default tasks.
  grunt.registerTask('default', ['cssmin']);
};
```

If you look at the UglifyJS example we did previously, you'll notice that they are very similar. Yay for standardization!

Gulp

At the time of writing this book, gulp.js[18] is the new kid on the block. Just like Grunt, Gulp is a task runner. They basically do the same thing. Some of the key differences are:

- Gulp calls itself 'The streaming build system'. It harnesses Node's streams to greatly increase build times. For more on streams, check out the stream handbook[19].
- Gulp has a simpler API. It has you writing JavaScript code instead of writing configurations in Grunt.
- Gulp plugins are written for one specific task only, whereas Grunt plugins can often do multiple things. You can find a searchable list of Gulp plugins here[20].

Installing Gulp

Installing Gulp is fairly simple using npm. Let's install Gulp globally.

```
npm install gulp -g
```

Next, navigate to your project root folder (or create one if you haven't already) and make sure you have a `package.json` set up. Don't forget, you can do this by running `npm init` or by referring to the Dependency Management chapter of this book. Then, run the following command:

[18]http://gulpjs.com/

[19]https://github.com/substack/stream-handbook

[20]http://gulpjs.com/plugins/

```
npm install gulp --save-dev
```

This will also install Gulp but this time locally. The important part is that we make a reference to it in the `devDependencies` section of `package.json` by using the `--save-dev` parameter.

Why did we install Gulp both globally and locally? Well, we first install Gulp globally so that we can access the `gulp` command anywhere from the command line. However, this can make deployment difficult because there is no local package installed in the project repository. We install Gulp locally to create a dependency on it in our `package.json`. It also allows you to install different versions of Gulp simultaneously. It's worth noting that when you run `gulp` in the project directory, the locally installed version will be used.

Example Usage

To demonstrate how to use Gulp, let's set up a task to uglify JavaScript files using the gulp-uglify[21] plugin. We'll start by installing the pluign:

```
npm install gulp-uglify --save-dev
```

Just as with Grunt, Gulp also has a single file for defining tasks. We'll create a file called `gulpfile.js` in the root of our project and use something like the following snippet to get started:

```
var gulp = require('gulp'),
        uglify = require('gulp-uglify');

// This task uglifies our JavaScript files.
gulp.task('compress', function() {
  gulp.src('src/js/*.js')
    .pipe(uglify())
    .pipe(gulp.dest('dist'))
});
```

You'll want to create a few dummy JavaScript files in the `src/js` folder to test with. From the command line, run `gulp compress` to run the `compress` task that you defined. You should then see the following status messages:

[21] https://www.npmjs.org/package/gulp-uglify

```
[11:36:18] Using gulpfile ~/myApp/gulpfile.js
[11:36:18] Starting 'compress'...
[11:36:18] Finished 'compress' after 8.87 ms
```

If you look in the `dist` folder, you'll notice that each of the JavaScript files we created have been uglified but unlike the Grunt plugin they have *not* been concatenated. This is because Gulp maintains the ideal that each plugin should only do one thing. So if we wanted to concatenate first and then uglify we would firstly need to install the gulp-concat[22] plugin:

```
npm install gulp-concat --save-dev
```

Then, we could update our `gulpfile.js` to look something like this:

```
var gulp = require('gulp'),
  uglify = require('gulp-uglify'),
  concat = require('gulp-concat');

// This task concatenates and uglifies our JavaScript files.
gulp.task('compress', function() {
  gulp.src('src/js/*.js')
    .pipe(concat('scripts.js'))
    .pipe(gulp.dest('dist'))
    .pipe(uglify())
    .pipe(gulp.dest('dist'))
});
```

Now, once you run `gulp compress`, you will end up with a single concatenated and uglified file at `dist/scripts.js`.

Before moving on from Gulp, we need to know how to set up file watching. Grunt required us to download a separate plugin to do this, whereas Gulp includes watch functionality from the get-go. Let's say we want to watch for any changes to our JavaScript files and run the `compress` task once that occurs. We would need to alter our `gulpfile.js` as shown below:

[22]https://www.npmjs.org/package/gulp-concat

```javascript
var gulp = require('gulp'),
  uglify = require('gulp-uglify'),
  concat = require('gulp-concat');

// This task concatenates and uglifies our JavaScript files.
gulp.task('compress', function() {
  gulp.src('src/js/*.js')
    .pipe(concat('scripts.js'))
    .pipe(gulp.dest('dist'))
    .pipe(uglify())
    .pipe(gulp.dest('dist'))
});

// Watch our files for changes.
gulp.task('watch', function() {
  gulp.watch('src/js/*.js', ['compress']);
});

// The default task is used when `gulp` is called from the command line.
gulp.task('default', ['compress', 'watch']);
```

We've set up a new task called `watch` which is fairly self-explanatory. Also, to simplify things further, we have defined another new task called `default` which runs our other tasks. So all we need to do now is hop over to the command line and run `gulp`. Now, whenever we make any changes to our JavaScript files, the `compress` task will run automagically.

Gulp is simple and fast. Once you learn its API, the idea of code over config is a very welcome one. We've only touched the surface of how powerful Gulp can be. For more information, read the official documentation[23].

So which one should you be using? Probably Grunt. Gulp is still on the bleeding edge. Its ecosystem needs more time before it should be considered as a replacement for Grunt. That being said, Gulp is an excellent tool and hopefully will continue to develop.

CodeKit

[23] https://github.com/gulpjs/gulp/blob/master/docs/README.md

CodeKit[24] is a Mac app created by Bryan Jones. It aims to take a lot of the pain out of setting up an automated workflow by providing an array of the most common and useful features for web developers:

- Has a built in web server which supports live browser refreshing across all of your devices.
- Watches for code file changes and compiles them with linting and minification.
- Has built in Bower support for installing and managing components.

The most appealing part of CodeKit is that it is very easy to get up and running. All you need to do is install the app and it just works. There is no fumbling through configuration files trying to figure out why your build script isn't working correctly. On top of that, the app's UI is cleanly designed and a pleasure to use.

Remember though that CodeKit is a Mac only app, so if you are working with other developers you may want to check they can actually run it before diving in.

To find out more, watch these demonstration videos[25] on the CodeKit homepage. Although there is a free trial, the app requires you to purchase a one time license[26] for full use.

Yeoman

Yeoman[27] defines a powerful workflow that helps you kickstart new projects by installing and configuring almost everything you could need to start developing your web applications. It greatly reduces the amount of time you need to write boilerplate code so that you can start developing sooner.

Yeoman provides an ecosystem of 'generators' which can be used for setting up the foundations of different types of applications. The generators are opinionated, meaning that although best standards are followed for the most part, they may not be perfectly configured for your specific needs.

There is a reason for leaving Yeoman until last. It can be quite tempting to use a generator to lay the scaffolding for your application without actually knowing everything that it is doing. Ideally, you should learn at least the basics of all the different tools that Yeoman installs and configures

[24]http://incident57.com/codekit/
[25]http://incident57.com/codekit/videos.html
[26]https://incident57.com/codekit/purchase.php
[27]http://yeoman.io/

before actually using Yeoman. That way, it is much easier for you to configure your application after Yeoman has done its magic.

The Yeoman workflow is comprised of three separately maintained tools: Yo, Bower and Grunt. Yo is the name of the actual scaffolding app from Yeoman, Bower is covered in the Dependency Management chapter of this book and Grunt has been covered at the beginning of this chapter.

Installing Yo

Yo[28] offers web application scaffolding, utilizing scaffolding templates referred to as generators. All we need is npm to install Yo:

```
npm install yo -g
```

Once Yo is installed, you will need to pick a generator best suited to your requirements. As an example let's install generator-angular[29], which utilizes a bunch of features that may be required of a typical AngularJS front-end web application workflow:

```
npm install generator-angular -g
```

As is the case with Yo, we install generators globally so that they can be accessed from anywhere in the command line. Once the generator is installed, we can create a new directory for our project and from within it run:

```
yo angular
```

You will be prompted with a few questions which will allow you to configure which tools will be installed. Once answered, sit back and relax as Yo takes care of downloading, installing and configuring everything. There is a lot of text scroll during this process so make sure your terminal window is full screen so that people around you think you're working in the matrix.

Once complete, you'll want to spend some time looking through all of the files and folders that have now been placed in your project root. In this particular generator, a whole bunch of tasks have been defined in your `Gruntfile.js` from which you can learn a lot about some of the best practices used in the developer community.

There are literally hundreds of generators available, many of which are unofficial and contributed by the community. You can view the list over at the Yeoman official page[30].

[28] https://www.npmjs.org/package/yo
[29] https://github.com/yeoman/generator-angular
[30] http://yeoman.io/generators/

Other Tips

The goal of this chapter is to help you become the most efficient developer you can be. To help you on your way, here are a few tips and tricks to make your life a little easier.

z

The more projects you work on, the more frustrating it can be to navigate between all of their directories in the command line. Enter 'z', a handy bash script that helps you to jump around directories in the command line. It tracks your most frequently used directories and allows you to navigate them by simply entering part of their name. For example:

```
z myApp
```

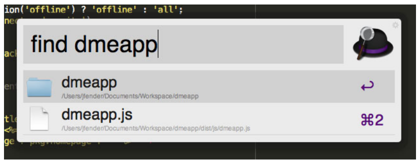
For example, this may take you to `/var/www/myApp` if that is a directory that you frequently access.

To install, grab the `z.sh` file from https://github.com/rupa/z[31] and write something like this in your `$HOME/.bashrc` or `$HOME/.zshrc`:

```
. /path/to/z.sh
```

Alfred.app

Alfred[32], perhaps named after the famous butler of Batman, is a Mac app that can be used as a replacement for the OS X Spotlight. Among many other useful productivity tasks, Alfred helps you save time by letting you search the web or your computer for files and applications with a few keystrokes. Alfred comes highly recommended to any hardcore Mac user. However, it is worth noting that Apple made some great changes[33] to the OS X Spotlight in Yosemite and you may find that sufficient enough.

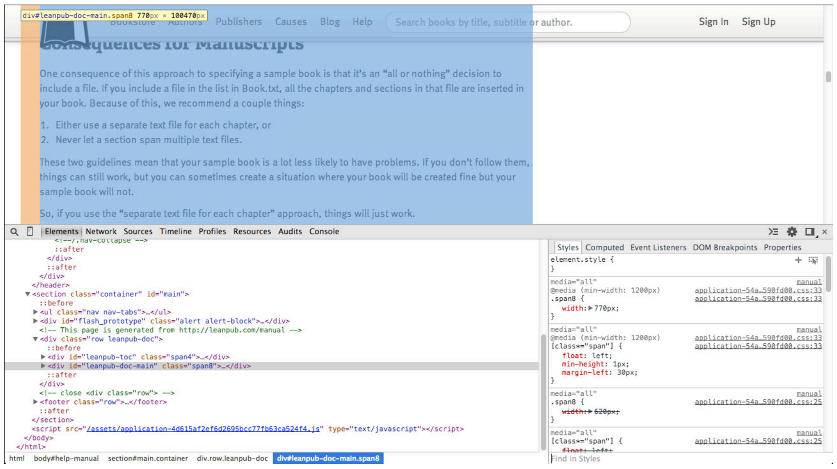


Alfred boosts productivity

[31]https://github.com/rupa/z
[32]http://www.alfredapp.com/
[33]http://www.macworld.com/article/2369722/hands-on-with-os-x-yosemite-spotlight-takes-center-stage.html

Chrome DevTools

One of the strongest attributes of the Google Chrome browser is its set of Developer Tools - more commonly known as DevTools. They are a set of web debugging tools directly accessible from within your browser. Although other browsers, such as Firefox, do come complete with their own set of tools for developers, Chrome probably provides the best implementation. If you've not used DevTools before, go ahead and install Chrome and read the overview documentation[34].



One consequence of this approach to specifying a sample book is that it's an "all or nothing" decision to include a file. If you include a file in the list in Book.txt, all the chapters and sections in that file are inserted in your book. Because of this, we recommend a couple things:

1. Either use a separate text file for each chapter, or
2. Never let a section span multiple text files.

These two guidelines mean that your sample book is a lot less likely to have problems. If you don't follow them, things can still work, but you can sometimes create a situation where your book will be created fine but your sample book will not.

So, if you use the "separate text file for each chapter" approach, things will just work.

```
        ::after
      </div>
      ::after
    </div>
  </header>
▼<section class="container" id="main">
    ::before
  ▶<ul class="nav nav-tabs">…</ul>
  ▶<div id="flash_prototype" class="alert alert-block">…</div>
    <!-- This page is generated from http://leanpub.com/manual -->
  ▼<div class="row leanpub-doc">
      ::before
    ▶<div id="leanpub-toc" class="span4">…</div>
    ▶<div id="leanpub-doc-main" class="span8">…</div>
      ::after
    </div>
    <!-- close <div class="row"> -->
  ▶<footer class="row">…</footer>
    ::after
  </section>
  <script src="/assets/application-4d615af2ef6d2695bcc77fb63ca524f4.js" type="text/javascript"></script>
</body>
</html>
```

html body#help-manual section#main.container div.row.leanpub-doc div#leanpub-doc-main.span8

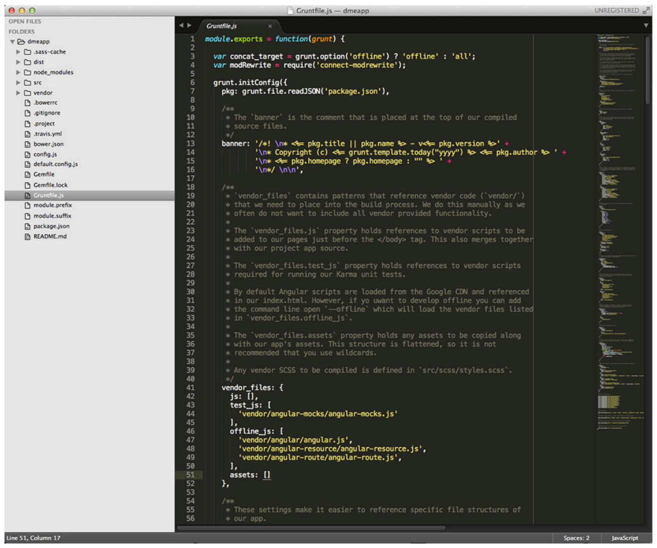
```
element.style {
}
media="all"                                   manual
@media (min-width: 1200px)    application-54a…590fd00.css:33
.span8 {                      application-54a…590fd00.css:33
  width:▶770px;
}
media="all"                                   manual
@media (min-width: 1200px)    application-54a…590fd00.css:33
[class*="span"] {             application-54a…590fd00.css:33
  float: left;
  min-height: 1px;
  margin-left: 30px;
}
media="all"                                   manual
.span8 {                      application-54a…590fd00.css:25
  ~~width:▶620px;~~
}
media="all"                                   manual
[class*="span"] {             application-54a…590fd00.css:25
  ~~float: left;~~
```

Chrome DevTools is the essential web page debugging tool

Sublime Text

Looking for a new text editor? Something with a super slick interface and huge range of features? Look no further. Sublime Text for Mac is quickly becoming one of the most popular text editors for developers.

[34]https://developer.chrome.com/devtools



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

  var concat_target = grunt.option('offline') ? 'offline' : 'all';
  var modRewrite = require('connect-modrewrite');

  grunt.initConfig({
    pkg: grunt.file.readJSON('package.json'),

    /**
     * The `banner` is the comment that is placed at the top of our compiled
     * source files.
     */
    banner: '/*! \n* <%= pkg.title || pkg.name %> - v<%= pkg.version %>' +
            '\n* Copyright (c) <%= grunt.template.today("yyyy") %> <%= pkg.author %> ' +
            '\n* <%= pkg.homepage ? pkg.homepage : "" %> ' +
            '\n*/ \n\n',

    /**
     * `vendor_files` contains patterns that reference vendor code (`vendor/`)
     * that we need to place into the build process. We do this manually as we
     * often do not want to include all vendor provided functionality.
     *
     * The `vendor_files.js` property holds references to vendor scripts to be
     * added to our pages just before the </body> tag. This also merges together
     * with our project app source.
     *
     * The `vendor_files.test_js` property holds references to vendor scripts
     * required for running our Karma unit tests.
     *
     * By default Angular scripts are loaded from the Google CDN and referenced
     * in our index.html. However, if yo uwant to develop offline you can add
     * the command line open `--offline` which will load the vendor files listed
     * in `vendor_files.offline_js`.
     *
     * The `vendor_files.assets` property holds any assets to be copied along
     * with our app's assets. This structure is flattened, so it is not
     * recommended that you use wildcards.
     *
     * Any vendor SCSS to be compiled is defined in `src/scss/styles.scss`.
     */
    vendor_files: {
      js: [],
      test_js: [
        'vendor/angular-mocks/angular-mocks.js'
      ],
      offline_js: [
        'vendor/angular/angular.js',
        'vendor/angular-resource/angular-resource.js',
        'vendor/angular-route/angular-route.js',
      ],
      assets: []
    },

    /**
     * These settings make it easier to reference specific file structures of
     * our app.
```

Sublime Text has a super slick interface

Its flagship feature, 'Goto Anything', has quickly become one of the main reasons for developers falling in love with Sublime Text. By simply pressing Command-P you can quickly navigate to any file in your project. For a full list of its features, and guidelines on how to install, check out the official homepage[35].

Summary

In this chapter, we took a fairy detailed look at the task runner Grunt and touched on other options such as Gulp and CodeKit. You've seen how important task runners can be, and how they can help you become a more efficient developer. We then looked at a few tips and tricks to speed up your workflow when developing.

[35]http://www.sublimetext.com/