SURVIVEJS

Webpack

FROM APPRENTICE TO MASTER

Juho Vepsäläinen

SurviveJS - Webpack

From apprentice to master

Juho Vepsäläinen, Tobias Koppers and Jesús Rodríguez Rodríguez

This book is for sale at http://leanpub.com/survivejs-webpack

This version was published on 2016-07-29

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

Introduction

Webpack[1] simplifies web development by solving a fundamental problem - the problem of bundling. It takes in various assets, such as JavaScript, CSS, and HTML, and then transforms these assets into a format that's easy to consume through a browser. By doing this well, it takes away significant amount of pain from web development.

It isn't the easiest tool to learn due to its configuration driven approach. The purpose of this guide is to help you get started with Webpack and then go beyond basics.

What is Webpack?

Web browsers have been designed to consume HTML, JavaScript, and CSS. The simplest way to develop is simply to write files that the browser understands directly. The problem is that this becomes unwieldy eventually. This is particularly true when you are developing web applications.

The naïve way to load JavaScript is simply to bundle it all into a single file. Eventually this won't be enough. You will need to start to split it up to benefit from caching. You can even begin to load dependencies dynamically as you need them. As an application develops, the complexity of handling it grows.

Webpack was developed to counter this problem. It handles all the aforementioned problems. It is possible to achieve the same results using different tools and build the processing workflows you need. In fact, that can be often enough. Task runners, such as Grunt and Gulp, allow you to achieve this, but even then you need to write a lot of configuration by hand.

How Webpack Changes the Situation?

Webpack takes another route. It allows you to treat your project as a dependency graph. You could have an *index.js* in your project that pulls in the dependencies the project needs through standard `import` statements. You can refer to your style files and other assets the same way.

Webpack does all the preprocessing for you and gives you the bundles you specify through configuration. This declarative approach is powerful, but it is a little difficult to learn. However, once you begin to understand how Webpack works, it becomes an indispensable tool. This book has been designed to get through that initial learning curve.

[1] https://webpack.github.io/

What Will You Learn?

This book has been designed to complement the official documentation of Webpack[2]. Even though the official documentation covers a lot of material, it might not be the easiest starting point for learning to use the tool. The goal of this book is to ease the learning curve while giving food for thought to more advanced users.

You will learn to develop a basic Webpack configuration for both development and production purposes. You will also learn advanced techniques that allow you to benefit from some of the greatest features of Webpack.

How is This Book Organized?

The first two parts of the book introduce you to Webpack and its basic concepts. You will develop basic configuration you can then expand to fit your specific purposes. The early chapters are task oriented. You can refer to them later if you forget how to achieve something specific with Webpack.

The latter portion of the book focuses more on advanced topics. You will learn more about loading specific types of assets. You will also deepen your understanding of Webpack's chunking mechanism, learn to utilize it for package authoring, and write your own loaders.

Who is This Book for?

I expect that you have a basic knowledge of JavaScript and Node.js. You should be able to use npm on an elementary level. If you know something about Webpack, that's great. By reading this book you will deepen your understanding of these tools.

If you happen to know Webpack well, there still might be something in the book for you. Skim through it and see if you can pick up some techniques. I've done my best to cover even some of the knottier parts of the tool.

If you find yourself struggling, consider seeking help from the community around the book. In case you are stuck or don't understand something, we are there to help. Any comments you might have will go towards improving the book content.

How to Approach the Book?

If you don't know much about the topic, consider going carefully through the first two parts. You can skim the rest to pick the bits you find interesting. If you know Webpack already, skim and pick up the techniques you find valuable.

[2] https://webpack.github.io/docs/

Book Versioning

Given this book receives a fair amount of maintenance and improvements due to the pace of innovation, there's a rough versioning scheme in place. I maintain release notes for each new version at the book blog[3]. That should give you a good idea of what has changed between versions. Also examining the GitHub repository may be beneficial. I recommend using the GitHub *compare* tool for this purpose. Example:

```
https://github.com/survivejs/webpack/compare/v1.2.0...v1.3.2
```

The page will show you the individual commits that went to the project between the given version range. You can also see the lines that have changed in the book. This excludes the private chapters, but it's enough to give you a good idea of the major changes made to the book.

The current version of the book is **1.3.2**.

The book is an on-going effort and I welcome feedback through various channels discussed below. I expand the guide based on demand to make it serve you as well as I can. You can even contribute fixes of your own to the book as the source is available.

A part of the profit goes towards funding the development of the tool itself. It also enables this type of work.

Getting Support

As no book is perfect, you will likely come by issues and might have some questions related to the content. There are a couple of options to deal with this:

- Contact me through GitHub Issue Tracker[4]
- Join me at Gitter Chat[5]
- Follow @survivejs[6] at Twitter for official news or poke me through @bebraw[7] directly
- Send me email at info@survivejs.com[8]
- Ask me anything about Webpack or React at SurviveJS AmA[9]

If you post questions to Stack Overflow, tag them using **survivejs** so I will get notified of them. You can use the hashtag **#survivejs** at Twitter for same effect.

[3] http://survivejs.com/blog/
[4] https://github.com/survivejs/webpack/issues
[5] https://gitter.im/survivejs/webpack
[6] https://twitter.com/survivejs
[7] https://twitter.com/bebraw
[8] mailto:info@survivejs.com
[9] https://github.com/survivejs/ama/issues

Announcements

I announce SurviveJS related news through a couple of channels:

- Mailing list[10]
- Twitter[11]
- Blog RSS[12]

Feel free to subscribe.

Acknowledgments

Big thanks to Christian Alfoni[13] for helping me to craft the first version of this book. That inspired the whole SurviveJS effort. The version you see now has been heavily revised.

The book wouldn't be half as good as it is without patient editing and feedback by my editor Jesús Rodríguez Rodríguez[14]. Thank you.

This book wouldn't have been possible without the original "SurviveJS - Webpack and React" effort. Anyone who contributed to that, deserves my thanks! You can check that book for more specific attributions.

Thanks to Mike "Pomax" Kamermans, Cesar Andreu, Dan Palmer, Viktor Jančík, Tom Byrer, Christian Hettlage, David A. Lee, Alexandar Castaneda, Marcel Olszewski, Steve Schwartz, Chris Sanders, Charles Ju, Aditya Bhardwaj, Rasheed Bustamam, José Menor, Ben Gale, and many others who have contributed direct feedback for this book!

[10] http://eepurl.com/bth1v5
[11] https://twitter.com/survivejs
[12] http://survivejs.com/atom.xml
[13] http://www.christianalfoni.com/
[14] https://github.com/Foxandxss

Webpack Compared

You can understand better why Webpack's approach is powerful by putting it into a historical context. Back in the day, it was enough just to concatenate some scripts together. Times have changed, though, and now distributing your JavaScript code can be a complex endeavor.

This problem has escalated with the rise of single page applications (SPAs). They tend to rely on numerous hefty libraries. There are multiple strategies on how to deal with loading them. You could load them all at once. You could also consider loading libraries as you need them. There are flexible solutions like this, and Webpack supports many of them.

The popularity of Node.js and npm[15], the Node.js package manager, provides more context. Before npm it was difficult to consume dependencies. Now that npm has become popular for front-end development, the situation has changed. Dependency management is far easier than earlier.

Task Runners and Bundlers

Historically speaking, there have been many build systems. Make is perhaps the best known, and is still a viable option. To make things easier, specialized *task runners*, such as Grunt, and Gulp appeared. Plugins available through npm made both task runners powerful.

Task runners are great tools on a high level. They allow you to perform operations in a cross-platform manner. The problems begin when you need to splice various assets together and produce bundles. This is the reason we have *bundlers*, such as Browserify, Brunch, or Webpack.

Continuing further on this path, JSPM[16] pushes package management directly to the browser. It relies on System.js[17], a dynamic module loader. Webpack 2 will support System.js semantics as well.

Unlike Browserify, Brunch, and Webpack, JSPM skips the bundling step altogether during development. You can generate a production bundle using it, however. Glen Maddern goes into good detail at his video about JSPM[18].

Make

You could say Make[19] goes way back. It was initially released in 1977. Even though it's an old tool, it has remained relevant. Make allows you to write separate tasks for various purposes. For instance,

[15] https://www.npmjs.com/
[16] http://jspm.io/
[17] https://github.com/systemjs/systemjs
[18] https://www.youtube.com/watch?t=33&v=iukBMY4apvI
[19] https://en.wikipedia.org/wiki/Make_%28software%29

you might have separate tasks for creating a production build, minifying your JavaScript or running tests. You can find the same idea in many other tools.

Even though Make is mostly used with C projects, it's not tied to it in any way. James Coglan discusses in detail how to use Make with JavaScript[20]. Consider the abbreviated code based on James' post below:

Makefile

```
PATH  := node_modules/.bin:$(PATH)
SHELL := /bin/bash

source_files := $(wildcard lib/*.coffee)
build_files  := $(source_files:%.coffee=build/%.js)
app_bundle   := build/app.js
spec_coffee  := $(wildcard spec/*.coffee)
spec_js      := $(spec_coffee:%.coffee=build/%.js)

libraries    := vendor/jquery.js

.PHONY: all clean test

all: $(app_bundle)

build/%.js: %.coffee
    coffee -co $(dir $@) $<

$(app_bundle): $(libraries) $(build_files)
    uglifyjs -cmo $@ $^

test: $(app_bundle) $(spec_js)
    phantomjs phantom.js

clean:
    rm -rf build
```

With Make, you model your tasks using Make-specific syntax and terminal commands. This allows it to integrate easily with Webpack.

[20] https://blog.jcoglan.com/2014/02/05/building-javascript-projects-with-make/

Grunt

Grunt

Grunt[21] went mainstream before Gulp. Its plugin architecture, especially, contributed towards its popularity. Often plugins are complex by themselves. As a result when configuration grows, it can become difficult to understand what's going on.

Here's an example from Grunt documentation[22]. In this configuration, we define a linting and a watcher task. When the *watch* task is run, it will trigger the *lint* task as well. This way, as we run Grunt, we'll get warnings in real-time in our terminal as we edit our source code.

Gruntfile.js

```
module.exports = function(grunt) {
  grunt.initConfig({
    lint: {
      files: ['Gruntfile.js', 'src/**/*.js', 'test/**/*.js'],
      options: {
        globals: {
          jQuery: true
        }
      }
    },
    watch: {
      files: ['<%= lint.files %>'],
      tasks: ['lint']
    }
```

[21] http://gruntjs.com/

[22] http://gruntjs.com/sample-gruntfile

```
  });

  grunt.loadNpmTasks('grunt-contrib-jshint');
  grunt.loadNpmTasks('grunt-contrib-watch');

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

In practice, you would have many small tasks like this for various purposes, such as building the project. An important part of the power of Grunt is that it hides a lot of the wiring from you. Taken too far, this can get problematic, though. It can become hard to thoroughly understand what's going on under the hood.

Note that the grunt-webpack[23] plugin allows you to use Webpack in a Grunt environment. You can leave the heavy lifting to Webpack.

Gulp

Gulp

Gulp[24] takes a different approach. Instead of relying on configuration per plugin, you deal with actual code. Gulp builds on top of the concept of piping. If you are familiar with Unix, it's the same idea here. You simply have sources, filters, and sinks.

Sources match to files. Filters perform operations on sources (e.g., convert to JavaScript). Finally, the results get passed to sinks (e.g., your build directory). Here's a sample *Gulpfile* to give you a better idea of the approach, taken from the project's README. It has been abbreviated a bit:

[23] https://www.npmjs.com/package/grunt-webpack
[24] http://gulpjs.com/

Gulpfile.js

```
var gulp = require('gulp');
var coffee = require('gulp-coffee');
var concat = require('gulp-concat');
var uglify = require('gulp-uglify');
var sourcemaps = require('gulp-sourcemaps');
var del = require('del');

var paths = {
    scripts: ['client/js/**/*.coffee', '!client/external/**/*.coffee']
};

// Not all tasks need to use streams
// A gulpfile is just another node program
// and you can use all packages available on npm
gulp.task('clean', function(cb) {
  // You can use multiple globbing patterns as
  // you would with `gulp.src`
  del(['build'], cb);
});

gulp.task('scripts', ['clean'], function() {
  // Minify and copy all JavaScript (except vendor scripts)
  // with sourcemaps all the way down
  return gulp.src(paths.scripts)
    .pipe(sourcemaps.init())
      .pipe(coffee())
      .pipe(uglify())
      .pipe(concat('all.min.js'))
    .pipe(sourcemaps.write())
    .pipe(gulp.dest('build/js'));
});

// Rerun the task when a file changes
gulp.task('watch', function() {
  gulp.watch(paths.scripts, ['scripts']);
});

// The default task (called when you run `gulp` from CLI)
gulp.task('default', ['watch', 'scripts']);
```

Given the configuration is code, you can always just hack it if you run into troubles. You can wrap

existing Node.js packages as Gulp plugins, and so on. Compared to Grunt, you have a clearer idea of what's going on. You still end up writing a lot of boilerplate for casual tasks, though. That is where some newer approaches come in.

gulp-webpack[25] allows you to use Webpack in a Gulp environment.

Fly[26] is a similar tool as Gulp. It relies on ES6 generators instead.

Browserify

Browserify

Dealing with JavaScript modules has always been a bit of a problem. The language itself actually didn't have the concept of modules till ES6. Ergo, we have been stuck in the '90s when it comes to browser environments. Various solutions, including AMD[27], have been proposed.

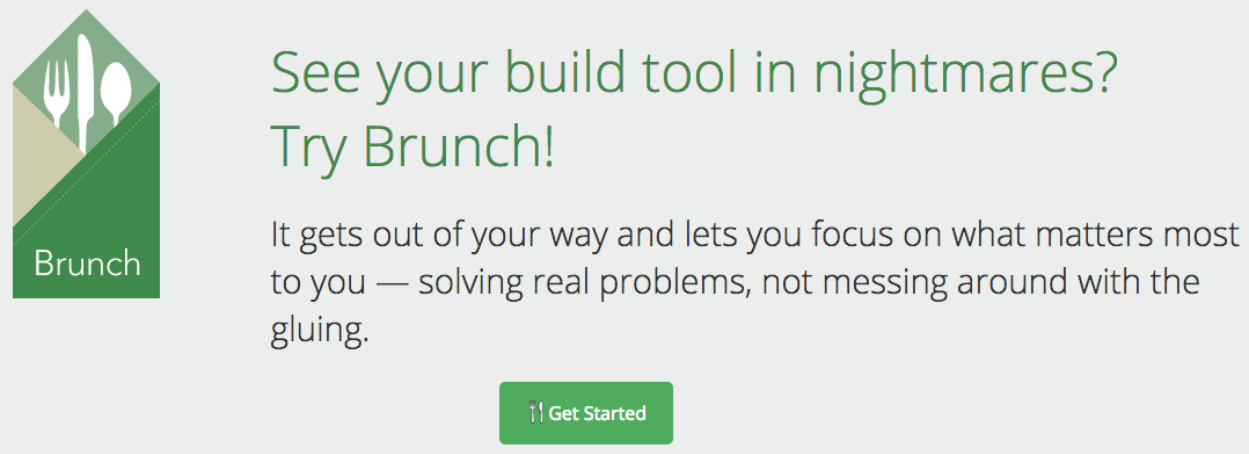
[25] https://www.npmjs.com/package/gulp-webpack
[26] https://github.com/bucaran/fly
[27] http://requirejs.org/docs/whyamd.html

In practice, it can be useful just to use CommonJS, the Node.js format, and let the tooling deal with the rest. The advantage is that you can often hook into npm and avoid reinventing the wheel.

Browserify[28] is one solution to the module problem. It provides a way to bundle CommonJS modules together. You can hook it up with Gulp. There are smaller transformation tools that allow you to move beyond the basic usage. For example, watchify[29] provides a file watcher that creates bundles for you during development. This will save some effort and no doubt is a good solution up to a point.

The Browserify ecosystem is composed of a lot of small modules. In this way, Browserify adheres to the Unix philosophy. Browserify is a little easier to adopt than Webpack, and is, in fact, a good alternative to it.

Brunch



Brunch

Compared to Gulp, Brunch[30] operates on a higher level of abstraction. It uses a declarative approach similar to Webpack's. To give you an example, consider the following configuration adapted from the Brunch site:

[28] http://browserify.org/
[29] https://www.npmjs.com/package/watchify
[30] http://brunch.io/

```
module.exports = {
  files: {
    javascripts: {
      joinTo: {
        'vendor.js': /^(?!app)/,
        'app.js': /^app/
      }
    },
    stylesheets: {
      joinTo: 'app.css'
    }
  },
  plugins: {
    babel: {
      presets: ['es2015', 'react']
    },
    postcss: {
      processors: [require('autoprefixer')]
    }
  }
};
```

Brunch comes with commands like `brunch new`, `brunch watch --server`, and `brunch build --production`. It contains a lot out of box and can be extended using plugins.

There is an experimental Hot Module Reloading runtime[31] for Brunch.

[31] https://github.com/brunch/hmr-brunch

JSPM

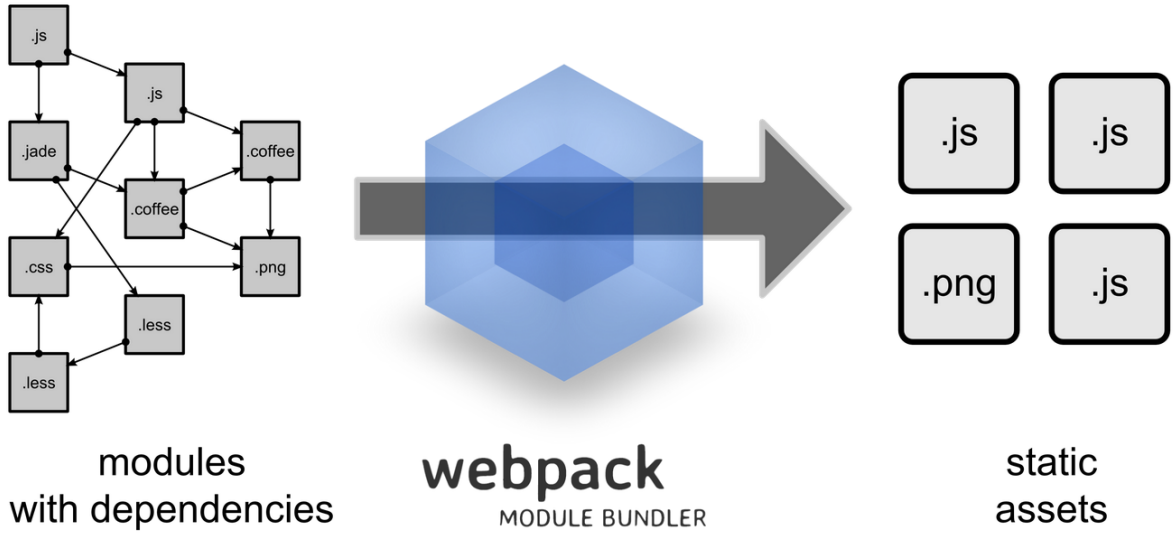
JSPM

Using JSPM is quite different than earlier tools. It comes with a little CLI tool of its own that is used to install new packages to the project, create a production bundle, and so on. It supports SystemJS plugins[32] that allow you to load various formats to your project.

Given JSPM is still a young project, there might be rough spots. That said, it may be worth a look if you are adventurous. As you know by now, tooling tends to change quite often in front-end development, and JSPM is definitely a worthy contender.

[32] https://github.com/systemjs/systemjs#plugins

Webpack



webpack

You could say Webpack[33] takes a more monolithic approach than Browserify. Whereas Browserify consists of multiple small tools, Webpack comes with a core that provides a lot of functionality out of the box. The core can be extended using specific *loaders* and *plugins*.

Webpack will traverse through the `require` statements of your project and will generate the bundles you have defined. The loader mechanism works for CSS as well and `@import` is supported. There are also plugins for specific tasks, such as minification, localization, hot loading, and so on.

To give you an example, `require('style!css!./main.css')` loads the contents of *main.css* and processes it through CSS and style loaders from right to left. Given that declarations, such as this, tie the source code to Webpack, it is preferable to set up the loaders at Webpack configuration. Here is a sample configuration adapted from the official webpack tutorial[34]:

webpack.config.js

[33] https://webpack.github.io/

[34] http://webpack.github.io/docs/tutorials/getting-started/

```
var webpack = require('webpack');

module.exports = {
  entry: './entry.js',
  output: {
    path: __dirname,
    filename: 'bundle.js'
  },
  module: {
    loaders: [
      {
        test: /\.css$/,
        loaders: ['style', 'css']
      }
    ]
  },
  plugins: [
    new webpack.optimize.UglifyJsPlugin()
  ]
};
```

Given the configuration is written in JavaScript, it's quite malleable. As long as it's JavaScript, Webpack is fine with it.

The configuration model may make Webpack feel a bit opaque at times. It can be difficult to understand what it's doing. This is particularly true for more complicated cases. Covering those is one of the main reasons why this book exists.

Why Webpack?

Why would you use Webpack over tools like Gulp or Grunt? It's not an either-or proposition. Webpack deals with the difficult problem of bundling, but there's so much more. I picked up Webpack because of its support for **Hot Module Replacement** (HMR). This is a feature used by babel-plugin-react-transform[35]. I will show you later how to set it up.

Hot Module Replacement

You might be familiar with tools, such as LiveReload[36] or Browsersync[37], already. These tools refresh the browser automatically as you make changes. HMR takes things one step further. In the case of

[35] https://github.com/gaearon/babel-plugin-react-transform
[36] http://livereload.com/
[37] http://www.browsersync.io/

React, it allows the application to maintain its state. This sounds simple, but it makes a big difference in practice.

Note that HMR is available in Browserify via livereactload[38], so it's not a feature that's exclusive to Webpack.

Bundle Splitting

Aside from the HMR feature, Webpack's bundling capabilities are extensive. It allows you to split bundles in various ways. You can even load them dynamically as your application gets executed. This sort of lazy loading comes in handy, especially for larger applications. You can load dependencies as you need them.

Asset Hashing

With Webpack, you can easily inject a hash to each bundle name (e.g., *app.d587bbd6e38337f5accd.js*). This allows you to invalidate bundles on the client side as changes are made. Bundle splitting allows the client to reload only a small part of the data in the ideal case.

Loaders and Plugins

All these smaller features add up. Surprisingly, you can get many things done out of the box. And if you are missing something, there are loaders and plugins available that allow you to go further.

Webpack comes with a significant learning curve. Even still, it's a tool worth learning, given it saves so much time and effort over the long term. To get a better idea how it compares to some other tools, check out the official comparison[39].

Conclusion

I hope this chapter helped you understand why Webpack is a valuable tool worth learning. It solves a fair share of common web development problems. If you know it well, it will save a great deal of time.

In the following chapters we'll examine Webpack in more detail. You will learn to develop a basic development and build configuration. The later chapters delve into more advanced topics.

You can use Webpack with some other tools. It won't solve everything. It does solve the difficult problem of bundling, however. That's one less worry during development. Just using *package.json*, `scripts`, and Webpack takes you far, as we will see soon.

[38] https://github.com/milankinen/livereactload

[39] https://webpack.github.io/docs/comparison.html

I Developing with Webpack

Webpack is driven by configuration, normally named as *webpack.config.js*. That's the heart of it all. The configuration defines the inputs and the outputs of your project. Most importantly, it describes the types of transformations you perform. These transformations are defined using **loaders** and **plugins** each of which serves a purpose of its own.

In this part I will show you how to set up a basic development setup based on Webpack. You will learn a bit about the underlying concepts as well.

1. Getting Started

Make sure you are using a recent version of Node.js[1] installed. I recommend using at least the most recent LTS (Long-Term Support) version. Before going further, you should have `node` and `npm` commands available at your terminal.

The completed configuration is available at GitHub[2]. If you are unsure of something, refer there.

It is possible to get a more controlled environment by using a solution such as Vagrant[3] or nvm[4]. Especially Vagrant comes with a performance penalty as it relies on a virtual machine. Vagrant is particularly useful in a team environment, though, as it gives you a predictable environment to develop against.

Particularly older version of Node.js (e.g. 0.10) are problematic and require extra work, such as polyfilling `Promise` through `require('es6-promise').polyfill()`. This technique depends on the es6-promise[5] package.

1.1 Setting Up the Project

To get a starting point, we should create a directory for our project and set up a *package.json* there. npm uses that to manage project dependencies. Here are the basic commands:

```
mkdir webpack-demo
cd webpack-demo
npm init -y # -y generates *package.json*, skip for more control
```

You can tweak the generated *package.json* manually to make further changes to it. We'll be doing some changes through *npm* tool, but manual tweaks are acceptable. The official documentation explains various package.json options[6] in more detail.

You can set those `npm init` defaults at *~/.npmrc*.

[1] http://nodejs.org/
[2] https://github.com/survivejs-demos/webpack-demo
[3] https://www.vagrantup.com/
[4] https://www.npmjs.com/package/nvm
[5] https://www.npmjs.com/package/es6-promise
[6] https://docs.npmjs.com/files/package.json

1.2 Installing Webpack

Even though Webpack can be installed globally (`npm i webpack -g`), I recommend maintaining it as a dependency of your project. This will avoid issues as then you will have control over the exact version you are running.

The approach works nicely in **Continuous Integration** (CI) setups as well. A CI system can install your local dependencies, compile your project using them, and then push the result to a server.

To add Webpack to our project, execute

```
npm i webpack --save-dev # or just -D if you want to save typing
```

You should see Webpack at your *package.json* `devDependencies` section after this. In addition to installing the package locally below the *node_modules* directory, npm also generates an entry for the executable.

1.3 Executing Webpack

You can display the exact path of the executables using `npm bin`. Most likely it points at *./node_modules/.bin*. Try executing Webpack from there through terminal using `node_modules/.bin/webpack` or a similar command.

After executing, you should see a version, a link to the command line interface guide and a long list of options. We won't be using most of those, but it's good to know that this tool is packed with functionality, if nothing else.

```
webpack-demo $ node_modules/.bin/webpack
webpack 1.13.0
Usage: https://webpack.github.io/docs/cli.html

Options:
  --help, -h, -?
  --config
  --context
  --entry
...
  --display-cached-assets
  --display-reasons, --verbose, -v

Output filename not configured.
```

We can use `--save` and `--save-dev` to separate application and development dependencies. The former will install and write to *package.json* `dependencies` field whereas the latter will write to `devDependencies` instead.

1.4 Directory Structure

As projects with just *package.json* are boring, we should set up something more concrete. To get started, we can implement a little web site that loads some JavaScript which we then build using Webpack. After we progress a bit, we'll end up with a directory structure like this:

- app/
  - index.js
  - component.js
- build/
- package.json
- webpack.config.js

The idea is that we'll transform that *app/* to as a bundle below *build/*. To make this possible, we should set up the assets needed and *webpack.config.js* of course.

1.5 Setting Up Assets

As you never get tired of `Hello world`, we might as well model a variant of that. Set up a component like this:

app/component.js

```
module.exports = function () {
  var element = document.createElement('h1');

  element.innerHTML = 'Hello world';

  return element;
};
```

Next, we are going to need an entry point for our application. It will simply `require` our component and render it through the DOM:

app/index.js

```
var component = require('./component');

document.body.appendChild(component());
```

1.6 Setting Up Webpack Configuration

We'll need to tell Webpack how to deal with the assets we just set up. For this purpose we'll develop a *webpack.config.js* file. Webpack and its development server will be able to discover this file through convention.

To keep things simple to maintain, we'll be using html-webpack-plugin[7] to generate an *index.html* for our application. *html-webpack-plugin* wires up the generated assets with it. Install it in the project:

```
npm i html-webpack-plugin --save-dev
```

Here is the configuration to setup the plugin and generate a bundle in our build directory:

webpack.config.js

```
const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');

const PATHS = {
  app: path.join(__dirname, 'app'),
  build: path.join(__dirname, 'build')
};

module.exports = {
  // Entry accepts a path or an object of entries.
  // We'll be using the latter form given it's
  // convenient with more complex configurations.
  entry: {
    app: PATHS.app
  },
  output: {
    path: PATHS.build,
    filename: '[name].js'
  },
  plugins: [
    new HtmlWebpackPlugin({
```

[7] https://www.npmjs.com/package/html-webpack-plugin

```
      title: 'Webpack demo'
    })
  ]
};
```

The `entry` path could be given as a relative one. The context[8] field can be used to configure that lookup. Given plenty of places expect absolute paths, I prefer to use absolute paths everywhere to avoid confusion.

If you execute `node_modules/.bin/webpack`, you should see output:

```
Hash: 2a7a7bccea1741de9447
Version: webpack 1.13.0
Time: 813ms
     Asset       Size  Chunks             Chunk Names
    app.js    1.69 kB       0  [emitted]  app
index.html  157 bytes          [emitted]
   [0] ./app/index.js 80 bytes {0} [built]
   [1] ./app/component.js 136 bytes {0} [built]
Child html-webpack-plugin for "index.html":
        + 3 hidden modules
```

The output tells us a lot. I've annotated it below:

- `Hash: 2a7a7bccea1741de9447` - The hash of the build. You can use this to invalidate assets through `[hash]` placeholder. We'll discuss hashing in detail at the *Adding Hashes to Filenames* chapter.
- `Version: webpack 1.13.0` - Webpack version.
- `Time: 813ms` - Time it took to execute the build.
- `app.js 1.69 kB 0 [emitted] app` - Name of the generated asset, size, the ids of the **chunks** into which it is related, status information telling how it was generated, name of the chunk.
- `[0] ./app/index.js 80 bytes {0} [built]` - The id of the generated asset, name, size, entry chunk id, the way it was generated.
- `Child html-webpack-plugin for "index.html":` - This is plugin related output. In this case *html-webpack-plugin* is doing output of its own.
- `+ 3 hidden modules` - This tells you that Webpack is omitting some output, namely modules within `node_modules` and similar directories. You can run Webpack using `webpack --display-modules` to display this information. See Stack Overflow[9] for an expanded explanation.

[8] https://webpack.github.io/docs/configuration.html#context

[9] https://stackoverflow.com/questions/28858176/what-does-webpack-mean-by-xx-hidden-modules

Examine the output below `build/`. If you look closely, you can see the same ids within the source. To see the application running, open the `build/index.html` file directly through a browser. On OS X `open ./build/index.html` works.

It can be convenient to use a tool like *serve* (`npm i serve -g`) to serve the build directory. In this case, execute `serve` at the output directory and head to `localhost:3000` at your browser. You can configure the port through the `--port` parameter.

I like to use `path.join`, but `path.resolve` would be a good alternative. See the Node.js path API[10] for further details.

favicons-webpack-plugin[11] makes it easy to deal with favicons using Webpack. It is compatible with *html-webpack-plugin*.

1.7 Adding a Build Shortcut

Given executing `node_modules/.bin/webpack` is a little verbose, we should do something about it. npm and *package.json* double as a task runner with some configuration. Adjust it as follows:

package.json

```
...
"scripts": {
  "build": "webpack"
},
...
```

You can execute these scripts through *npm run*. For instance, in this case we could use *npm run build*. As a result you should get build output as before.

This works because npm adds *node_modules/.bin* temporarily to the path. As a result, rather than having to write `"build": "node_modules/.bin/webpack"`, we can do just `"build": "webpack"`.

There are shortcuts like *npm start* and *npm test*. We can run these directly without *npm run* although that will work too.

[10] https://nodejs.org/api/path.html

[11] https://www.npmjs.com/package/favicons-webpack-plugin

It is possible to execute *npm run* anywhere within the project. It doesn't have to be run in the project root in order to work.

1.8 Conclusion

Even though we've managed to set up a basic Webpack setup, it's not that great yet. Developing against it would be painful. Each time we wanted to check out our application, we would have to build it manually using `npm run build` and then refresh the browser.

That's where Webpack's more advanced features come in. To make room for these features, I will show you how to split your Webpack configuration in the next chapter.

2. Splitting the Configuration

At minimum, your Webpack configuration can be contained in a single file. As the needs of your project grow, you'll need to figure out means to manage it. It becomes necessary to split it up per environment so that you have enough control over the build result. There is no single right way to achieve this, but at least the following ways are feasible:

- Maintain configuration in multiple files and point Webpack to each through `--config` parameter. Share configuration through module imports. You can see this approach in action at webpack/react-starter[1].
- Push configuration to a library which you then consume. Example: HenrikJoreteg/hjs-webpack[2].
- Maintain configuration within a single file and branch there. If we trigger a script through *npm* (i.e., `npm run test`), npm sets this information in an environment variable. We can match against it and return the configuration we want.

I prefer the last approach as it allows me to understand what's going on easily. I've developed a little tool known as webpack-merge[3] to achieve this. I'll show you how to set it up next.

2.1 Setting Up *webpack-merge*

To get started, execute

```
npm i webpack-merge --save-dev
```

to add *webpack-merge* to the project.

Next, we need to define some split points to our configuration so we can customize it per npm script. Here's the basic idea:

webpack.config.js

[1] https://github.com/webpack/react-starter
[2] https://github.com/HenrikJoreteg/hjs-webpack
[3] https://www.npmjs.org/package/webpack-merge

```
const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');
const merge = require('webpack-merge');

const PATHS = {
  app: path.join(__dirname, 'app'),
  build: path.join(__dirname, 'build')
};

module.exports = {
const common = {
  // Entry accepts a path or an object of entries.
  // We'll be using the latter form given it's
  // convenient with more complex configurations.
  entry: {
    app: PATHS.app
  },
  output: {
    path: PATHS.build,
    filename: '[name].js'
  },
  plugins: [
    new HtmlWebpackPlugin({
      title: 'Webpack demo'
    })
  ]
};

var config;

// Detect how npm is run and branch based on that
switch(process.env.npm_lifecycle_event) {
  case 'build':
    config = merge(common, {});
    break;
  default:
    config = merge(common, {});
}

module.exports = config;
```

After this change our build should behave exactly the same way as before. This time, however, we have room for expansion. We can hook up **Hot Module Replacement** in the next chapter to make

the browser refresh and turn our the development mode into something more useful.

2.2 Integrating *webpack-validator*

To make it easier to develop our configuration, we can integrate a tool known as webpack-validator[4] to our project. It will validate the configuration against a schema and warn if we are trying to do something not sensible. This takes some pain out of learning and using Webpack.

Install it first:

```
npm i webpack-validator --save-dev
```

Integrating it to our project is straight-forward:

webpack.config.js

```
const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');
const merge = require('webpack-merge');
const validate = require('webpack-validator');

...

module.exports = config;
module.exports = validate(config);
```

If you break your Webpack configuration somehow after doing this, the validator will likely notice and give you a nice validation error to fix.

2.3 Conclusion

Even though a simple technique, splitting configuration this way makes room for growing your setup. Each approach comes with its pros and cons. I find this one works well for small to medium projects. Bigger projects might need a different approach.

Now that we have room for growth, I will show you how to set up automatic browser refresh with Webpack in the next chapter.

[4] https://www.npmjs.com/package/webpack-validator

3. Automatic Browser Refresh

Tools, such as LiveReload[1] or Browsersync[2], allow us to refresh the browser as we develop our application. They can even avoid refresh for CSS changes.

A good first step towards a better development environment would be to use Webpack in its **watch** mode. You can activate it through `webpack --watch`. Once enabled, it will detect changes made to your files and recompiles automatically. A solution known as *webpack-dev-server* builds on top of the watch mode and goes even further.

webpack-dev-server is a development server running in-memory. It refreshes content automatically in the browser while you develop your application. It also supports an advanced Webpack feature known as Hot Module Replacement (HMR), which provides a way to patch the browser state without a full refresh. This is particularly powerful with technology such as React.

You should use *webpack-dev-server* strictly for development. If you want to host your application, consider other standard solutions, such as Apache or Nginx.

An IDE feature known as **safe write** can wreak havoc with hot loading. Therefore it is advisable to turn it off when using a HMR based setup.

3.1 Getting Started with *webpack-dev-server*

To get started with *webpack-dev-server*, execute:

```
npm i webpack-dev-server --save-dev
```

As before, this command will generate a command below the `npm bin` directory. You could try running *webpack-dev-server* from there. The quickest way to enable automatic browser refresh for our project is to run `webpack-dev-server --inline`. `--inline`, runs the server in so called *inline* mode that writes the webpack-dev-server client to the resulting code.

Attaching *webpack-dev-server* to the Project

To integrate *webpack-dev-server* to our project, we can follow the same idea as in the previous chapter and define a new command to the `scripts` section of *package.json*:

package.json

[1] http://livereload.com/

[2] http://www.browsersync.io/

```
...
"scripts": {
  "start": "webpack-dev-server",
  "build": "webpack"
},
...
```

We'll add that `--inline` part back through Webpack configuration in a bit. I prefer to keep the npm *scripts* portion as simple as possible and push the complexity to configuration. Even though it's more code to write, it's also easier to maintain as you can see easier what's going on.

If you execute either *npm run start* or *npm start* now, you should see something like this at the terminal:

```
> webpack-dev-server

[webpack-validator] Config is valid.
http://localhost:8080/webpack-dev-server/
webpack result is served from /
content is served from .../webpack-demo
Hash: 2dca5a3850ce5d2de54c
Version: webpack 1.13.0
```

The output means that the development server is running. If you open *http://localhost:8080/* at your browser, you should see something.

If you try modifying the code, you should see output at your terminal. The problem is that the browser doesn't catch these changes without a hard refresh and that flag. That's something we need to resolve next through configuration.

Hello world

Hello world

If you fail to see anything at the browser, you may need to use a different port. The server might fail to run because there's something else running in the port. You can verify this through terminal using a command such as `netstat -na | grep 8080`. If there's something running in the port 8080, it should display a message. The exact command may depend on the platform.

3.2 Configuring Hot Module Replacement (HMR)

Hot Module Replacement builds on top the *webpack-dev-server*. It enables an interface that makes it possible to swap modules live. For example, *style-loader* is able to update your CSS without forcing a refresh. It is easy to perform HMR with CSS as it doesn't contain any application state.

HMR is possible with JavaScript too, but due to the state we have in our applications, it's harder. In the *Configuring React* chapter we discuss how to set it up with React. You can use the same idea elsewhere.

We could use `webpack-dev-server --inline --hot` to achieve this from the CLI. `--hot` enables the HMR portion from Webpack through a specific plugin designed for this purpose and writes an entry pointing to a JavaScript file related to it.

Defining Configuration for HMR

To keep our configuration manageable, I'll split functionalities like HMR into parts of their own. This keeps our *webpack.config.js* simple and promotes reuse. We could push a collection like this to a npm package of its own. We could even turn them into presets to use across projects. Functional composition allows that.

I'll push all of our configuration parts to *libs/parts.js* and consume them from there. Here's what a part would look like for HMR:

libs/parts.js

```
const webpack = require('webpack');

exports.devServer = function(options) {
  return {
    devServer: {
      // Enable history API fallback so HTML5 History API based
      // routing works. This is a good default that will come
      // in handy in more complicated setups.
      historyApiFallback: true,

      // Unlike the cli flag, this doesn't set
      // HotModuleReplacementPlugin!
      hot: true,
      inline: true,

      // Display only errors to reduce the amount of output.
      stats: 'errors-only',
```

```
      // Parse host and port from env to allow customization.
      //
      // If you use Vagrant or Cloud9, set
      // host: options.host || '0.0.0.0';
      //
      // 0.0.0.0 is available to all network devices
      // unlike default `localhost`.
      host: options.host, // Defaults to `localhost`
      port: options.port // Defaults to 8080
    },
    plugins: [
      // Enable multi-pass compilation for enhanced performance
      // in larger projects. Good default.
      new webpack.HotModuleReplacementPlugin({
        multiStep: true
      })
    ]
  };
}
```

It's plenty of code, but it's better to encapsulate it so it contains ideas we understand and want to reuse later on. Fortunately hooking up this part with our main configuration is simple:

webpack.config.js

```
const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');
const merge = require('webpack-merge');
const validate = require('webpack-validator');

const parts = require('./libs/parts');

...

// Detect how npm is run and branch based on that
switch(process.env.npm_lifecycle_event) {
  case 'build':
    config = merge(common, {});
    break;
  default:
    config = merge(common, {});
    config = merge(
      common,
```

```
      parts.devServer({
        // Customize host/port here if needed
        host: process.env.HOST,
        port: process.env.PORT
      })
    );
}

module.exports = validate(config);
```

Execute `npm start` and surf to **localhost:8080**. Try modifying *app/component.js*. It should refresh the browser. Note that this is a hard refresh in case you modify JavaScript code. CSS modifications work in a neater manner and can be applied without a refresh as we will see in the next chapter.

webpack-dev-server can be very particular about paths. If the given `include` paths don't match the system casing exactly, this can cause it to fail to work. Webpack issue #675[3] discusses this in more detail.

You should be able to access the application alternatively through **localhost:8080/webpack-dev-server/** instead of the root. It will provide status information within the browser itself at the top of the application.

HMR on Windows, Ubuntu, and Vagrant

The setup may be problematic on certain versions of Windows, Ubuntu, and Vagrant. We can solve this through polling:

libs/parts.js

```
const webpack = require('webpack');

exports.devServer = function(options) {
  return {
    watchOptions: {
      // Delay the rebuild after the first change
      aggregateTimeout: 300,
      // Poll using interval (in ms, accepts boolean too)
      poll: 1000
```

[3] https://github.com/webpack/webpack/issues/675

```
    },
    devServer: {
      ...
    },
    ...
  };
}
```

Given this setup polls the filesystem, it is going to be more resource intensive. It's worth giving a go if the default doesn't work, though.

There are more details in *webpack-dev-server* issue #155[4].

3.3 Accessing the Development Server from Network

It is possible to customize host and port settings through the environment in our setup (i.e., `export PORT=3000` on Unix or `SET PORT=3000` on Windows). This can be useful if you want to access your server using some other device within the same network. The default settings are enough on most platforms.

To access your server, you'll need to figure out the ip of your machine. On Unix this can be achieved using `ifconfig | grep inet`. On Windows `ipconfig` can be used. An npm package, such as node-ip[5] may come in handy as well. Especially on Windows you may need to set your `HOST` to match your ip to make it accessible.

3.4 Alternative Ways to Use *webpack-dev-server*

We could have passed *webpack-dev-server* options through terminal. I find it clearer to manage it within Webpack configuration as that helps to keep *package.json* nice and tidy. It is also easier to understand what's going on as you don't need to dig the answers from Webpack source.

Alternatively, we could have set up an Express server of our own and used *webpack-dev-server* as a middleware[6]. There's also a Node.js API[7]. This is a good approach if you want control and flexibility.

dotenv[8] allows you to define environment variables through a *.env* file. This can be somewhat convenient during development!

[4] https://github.com/webpack/webpack-dev-server/issues/155
[5] https://www.npmjs.com/package/node-ip
[6] https://webpack.github.io/docs/webpack-dev-middleware.html
[7] https://webpack.github.io/docs/webpack-dev-server.html#api
[8] https://www.npmjs.com/package/dotenv

Note that there are slight differences[9] between the CLI and the Node.js API. This is the reason why some prefer to solely use the Node.js API.

3.5 Conclusion

In this chapter you learned to set up Webpack to refresh your browser automatically. We can go a notch further and make this work beautifully with CSS files. We'll do that in the next chapter.

[9] https://github.com/webpack/webpack-dev-server/issues/106

4. Refreshing CSS

In this chapter we'll set up CSS with our project and see how it works out with automatic browser refreshing. The neat thing is that in this case Webpack doesn't have to force a full refresh. Instead, it can do something smarter as we'll see soon.

4.1 Loading CSS

To set up CSS, we'll need to use a couple of loaders. To get started, invoke

```
npm i css-loader style-loader --save-dev
```

Now that we have the loaders we need, we'll need to make sure Webpack is aware of them. Configure as follows:

libs/parts.js

```
...

exports.setupCSS = function(paths) {
  return {
    module: {
      loaders: [
        {
          test: /\.css$/,
          loaders: ['style', 'css'],
          include: paths
        }
      ]
    }
  };
}
```

We also need to connect our configuration fragment with the main configuration:

webpack.config.js

```
...

switch(process.env.npm_lifecycle_event) {
  case 'build':
    config = merge(common, {});
    config = merge(
      common,
      parts.setupCSS(PATHS.app)
    );
    break;
  default:
    config = merge(
      common,
      parts.setupCSS(PATHS.app),
      ...
    );
}

module.exports = validate(config);
```

The configuration we added means that files ending with `.css` should invoke given loaders. `test` matches against a JavaScript style regular expression. The loaders are evaluated from right to left.

In this case, *css-loader* gets evaluated first, then *style-loader*. *css-loader* will resolve `@import` and `url` statements in our CSS files. *style-loader* deals with `require` statements in our JavaScript. A similar approach works with CSS preprocessors, like Sass and Less, and their loaders.

Loaders are transformations that are applied to source files, and return the new source. Loaders can be chained together, like using a pipe in Unix. See Webpack's What are loaders?[1] and list of loaders[2].

If `include` isn't set, Webpack will traverse all files within the base directory. This can hurt performance! It is a good idea to set up `include` always. There's also `exclude` option that may come in handy. Prefer `include`, however.

4.2 Setting Up the Initial CSS

We are missing just one bit, the actual CSS itself:

app/main.css

[1] http://webpack.github.io/docs/using-loaders.html
[2] http://webpack.github.io/docs/list-of-loaders.html

```
body {
  background: cornsilk;
}
```

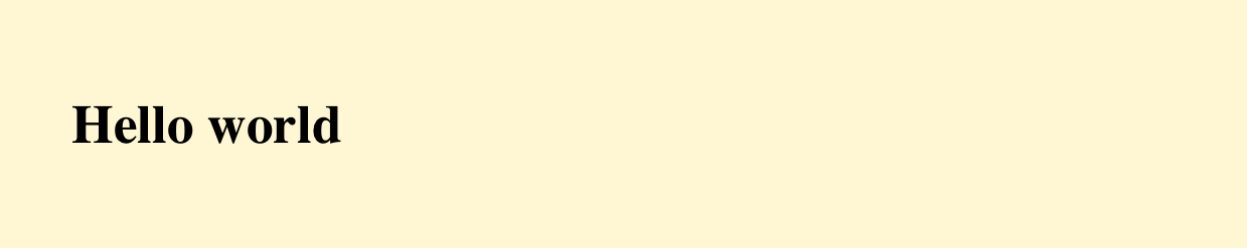
Also, we'll need to make Webpack aware of it. Without having a `require` pointing at it, Webpack won't be able to find the file:

app/index.js

```
require('./main.css');

...
```

Execute `npm start` now. Point your browser to **localhost:8080** if you are using the default port.

Open up *main.css* and change the background color to something like `lime` (`background: lime`). Develop styles as needed to make it look a little nicer.



Hello cornsilk world

An alternative way to load CSS would be to define a separate entry through which we point at CSS.

4.3 Understanding CSS Scoping and CSS Modules

When you `require` a CSS file like this, Webpack will include it to the bundle where you `require` it. Assuming you are using the *style-loader*, Webpack will write it to a `style` .tag. This means it's going to be in a global scope by default!

Specification known as CSS Modules[3] allows you to default to local scoping. Webpack's *css-loader* supports it. So if you want local scope by default over a global one, enable them through `css?modules`. After this you'll need to wrap your global styles within `:global(body) { ... }` kind of declarations.

[3] https://github.com/css-modules/css-modules

The query syntax, `css?modules` is discussed in greater detail in the *Loader Definitions* chapter. There are multiple ways to achieve the same effect in Webpack.

In this case the `require` statement will give you the local classes you can then bind to elements. Assuming we had styling like this:

app/main.css

```
:local(.redButton) {
  background: red;
}
```

We could then bind the resulting class to a component like this:

app/component.js

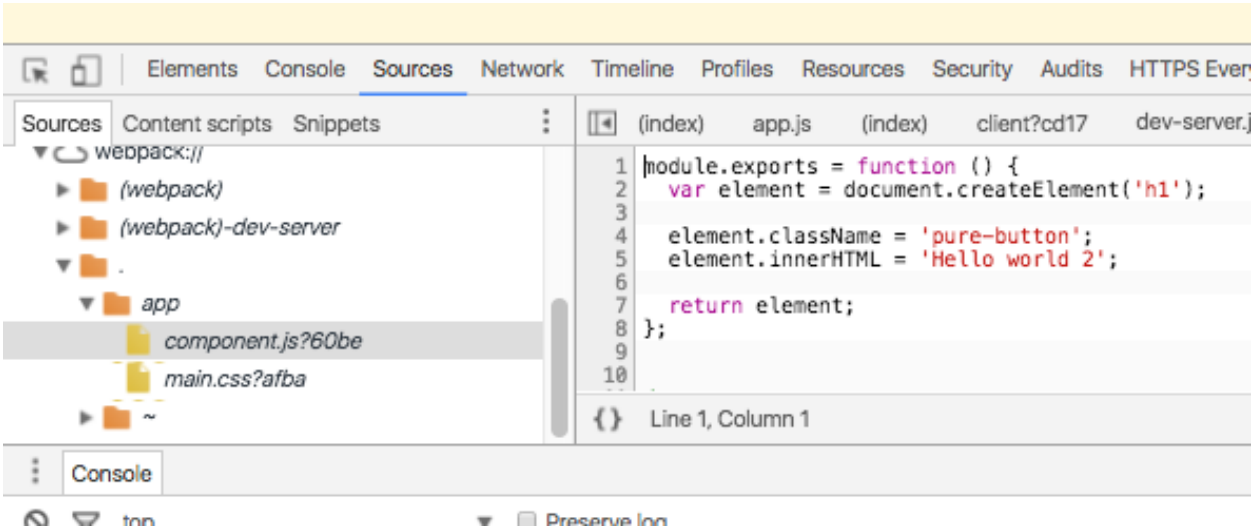
```
var styles = require('./main.css');

...

// Attach the generated class name
element.className = styles.redButton;
```

Even though this might feel like a strange way of working, defaulting to local scoping can take away a lot of pain you encounter with CSS. We'll be using old skool styling in this little demonstration project of ours, but it's a good technique to be aware of. It even enables features like composition so it's worth knowing.

4.4 Conclusion

In this chapter, you learned to set up Webpack to refresh your browser during development. The next chapter covers a convenience feature known as sourcemaps.

5. Enabling Sourcemaps



Sourcemaps in Chrome

To improve the debuggability of the application, we can set up sourcemaps for both code and styling. Sourcemaps allow you to see exactly where an error was raised. Webpack can generate both inline sourcemaps included within bundles or separate sourcemap files. The former is useful during development due to better performance while the latter is handy for production usage as it will keep the bundle size small.

I'll show you how to enable sourcemaps for JavaScript code next. It is a good idea to study the documentation of the loaders you are using to see specific tips. For example with TypeScript you may need to set a certain flag to make it to work.

5.1 Enabling Sourcemaps During Development

To enable sourcemaps during development, we can use a decent default known as `eval-source-map` and for production we can use normal sourcemaps (separate file) like this:

webpack.config.js

```
...

switch(process.env.npm_lifecycle_event) {
  case 'build':
    config = merge(
      common,
      {
        devtool: 'source-map'
      },
      parts.setupCSS(PATHS.app)
    );
  default:
    config = merge(
      common,
      {
        devtool: 'eval-source-map'
      },
      parts.setupCSS(PATHS.app),
      ...
    );
}

module.exports = validate(config);
```

`eval-source-map` builds slowly initially, but it provides fast rebuild speed and yields real files. Faster development specific options, such as `cheap-module-eval-source-map` and `eval`, produce lower quality sourcemaps. All `eval` options will emit sourcemaps as a part of your JavaScript code.

It is possible you may need to enable sourcemaps in your browser for this to work. See Chrome[1], Firefox[2], IE Edge[3], and Safari[4] instructions for further details.

5.2 Sourcemap Types Supported by Webpack

Even though a sourcemap type, such as `eval-source-map` or `eval`, can be enough during development, Webpack provides other types too. Given these will be generated within your bundles, they won't be useful during production.

[1] https://developer.chrome.com/devtools/docs/javascript-debugging

[2] https://developer.mozilla.org/en-US/docs/Tools/Debugger/How_to/Use_a_source_map

[3] https://developer.microsoft.com/en-us/microsoft-edge/platform/documentation/f12-devtools-guide/debugger/#source-maps

[4] https://developer.apple.com/library/safari/documentation/AppleApplications/Conceptual/Safari_Developer_Guide/ResourcesandtheDOM/ResourcesandtheDOM.html#//apple_ref/doc/uid/TP40007874-CH3-SW2

The following table adapted from the documentation[5] contains the supported types arranged based on speed. The lower the quality, the higher build and rebuild speeds are possible.

Sourcemap type	Quality	Notes
`eval`	Generated code	Each module is executed with `eval` and `//@ sourceURL`.
`cheap-eval-source-map`	Transformed code (lines only)	Each module is executed with `eval` and a sourcemap is added as a dataurl to the `eval`.
`cheap-module-eval-source-map`	Original source (lines only)	Same idea, but higher quality with lower performance.
`eval-source-map`	Original source	Same idea, but highest quality and lowest performance.

You can start from `eval-source-map` and move to other options as it starts to feel too slow.

Webpack can also generate production usage friendly sourcemaps. These will end up in separate files and will be loaded by the browser only when required. This way your users get good performance while it's easier for you to debug the application. I've listed them in a similar way below.

Sourcemap type	Quality	Notes
`cheap-source-map`	Transformed code (lines only)	Generated sourcemaps don't have column mappings. Sourcemaps from loaders are not used.
`cheap-module-source-map`	Original source (lines only)	Same except sourcemaps from loaders are simplified to a single mapping per line.
`source-map`	Original source	The best quality with the most complete result, but also the slowest.

`source-map` is a good default here. Even though it will take longer to generate the sourcemaps this way, you will get the best quality. If you don't care about production sourcemaps, you can simply skip the setting there and get better performance in return.

There are a couple of other options that affect sourcemap generation:

[5] https://webpack.github.io/docs/configuration.html#devtool

```
const config = {
  output: {
    // Modify the name of the generated sourcemap file.
    // You can use [file], [id], and [hash] replacements here.
    // The default option is enough for most use cases.
    sourceMapFilename: '[file].map', // Default

    // This is the sourcemap filename template. It's default format
    // depends on the devtool option used. You don't need to modify this
    // often.
    devtoolModuleFilenameTemplate: 'webpack:///[resource-path]?[loaders]'
  },
  ...
};
```

The official documentation[6] digs into devtool specifics.

5.3 `SourceMapDevToolPlugin`

If you want more control over sourcemap generation, it is possible to use the `SourceMapDevTool-Plugin` instead. This way you can generate sourcemaps only for the portions of the code you want while having strict control over the result. In case you use the plugin, you can skip `devtool` option altogether.

Here's what it looks like in its entirety (adapted from the official documentation[7]):

```
const config = {
  plugins: [
    new webpack.SourceMapDevToolPlugin({
      // Match assets just like for loaders.
      test: string | RegExp | Array,
      include: string | RegExp | Array,

      // `exclude` matches file names, not package names!
      exclude: string | RegExp | Array,

      // If filename is set, output to this file.
```

[6] https://webpack.github.io/docs/configuration.html#output-sourcemapfilename
[7] https://webpack.github.io/docs/list-of-plugins.html#sourcemapdevtoolplugin

```
      // See `sourceMapFileName`.
      filename: string,

      // This line is appended to the original asset processed. For
      // instance '[url]' would get replaced with an url to the
      // sourcemap.
      append: false | string,

      // See `devtoolModuleFilenameTemplate` for specifics.
      moduleFilenameTemplate: string,
      fallbackModuleFilenameTemplate: string,

      module: bool, // If false, separate sourcemaps aren't generated.
      columns: bool, // If false, column mappings are ignored.

      // Use simpler line to line mappings for the matched modules.
      lineToLine: bool | {test, include, exclude}
    }),
    ...
  ],
  ...
};
```

5.4 Sourcemaps for Styling

If you want to enable sourcemaps for styling files, you can achieve this using a query parameter. Loaders, such as *css-loader*, *sass-loader*, and *less-loader*, accept a `?sourceMap` (i.e, `css?sourceMap`).

This isn't without gotchas. The *css-loader* documentation notes that relative paths within CSS declarations are known to be buggy and suggests using setting an absolute public path (`output.publicPath`) resolving to the server url.

5.5 Conclusion

Sourcemaps can be convenient during development. They provide us better means to debug our applications as we can still examine the original code over generated one. They can be useful even for production usage and allow you to debug issues while serving a client friendly version of your application.

Our build configuration isn't that sophisticated yet, though. I'll show you how to push it further in the next part as we discuss various build related techniques.

II Building with Webpack

In the previous part I showed you how to set up a basic development setup with Webpack. In this part we'll dig into build related concerns. We'll discuss aspects like code minification, bundle splitting, and caching.

6. Minifying the Build

So far we haven't given thought to our build output and no doubt it's going to be a little chunky, especially as we include React in it. We can apply a variety of techniques to bring down the size of it. We can also leverage client level caching and load certain assets lazily. I'll discuss the latter topic in the *Understanding Chunks* chapter later on.

The first of these techniques is known as minification. It is a process where code is simplified without losing any meaning that matters to the interpreter. As a result your code will most likely look quite jumbled and it will be hard to read. But that's the point.

Even if we minify our build, we can still generate sourcemaps through the `devtool` option we discussed earlier. This will give us better means to debug even production code.

6.1 Generating a Baseline Build

To get started, we should generate a baseline build so we have something to optimize. Given our project is so small, there isn't much to optimize yet. We could bring a large dependency like React to get something to slim up. Install React first:

```
npm i react --save
```

We also need to make our project depend on it:

app/index.js

```
require('react');

...
```

Now that we have something to optimize, execute `npm run build`. You should end up with something like this:

```
[webpack-validator] Config is valid.
Hash: dde0419cac0674732a83
Version: webpack 1.13.0
Time: 1730ms
     Asset       Size  Chunks             Chunk Names
    app.js     133 kB       0  [emitted]  app
app.js.map     157 kB       0  [emitted]  app
index.html  157 bytes          [emitted]
   [0] ./app/index.js 123 bytes {0} [built]
  [37] ./app/component.js 136 bytes {0} [built]
    + 36 hidden modules
Child html-webpack-plugin for "index.html":
        + 3 hidden modules
```

133 kB is a lot! Minification should bring down the size a lot.

6.2 Minifying the Code

Minification will convert our code into a smaller format without losing any meaning. Usually this means some amount of rewriting code through predefined transformations. Sometimes, this can break code as it can rewrite pieces of code you inadvertently depend upon.

The easiest way to enable minification is to call `webpack -p`. `-p` is a shortcut for `--optimize-minimize`, you can think it as `-p` for "production". Alternatively, we can use a plugin directly as this provides us more control. By default Uglify will output a lot of warnings and they don't provide value in this case, we'll be disabling them.

As earlier, we can define a little function for this purpose and then point to it from our main configuration. Here's the basic idea:

libs/parts.js

```
...

exports.minify = function() {
  return {
    plugins: [
      new webpack.optimize.UglifyJsPlugin({
        compress: {
          warnings: false
        }
      })
    ]
```

```
  };
}
```

Now we can hook it up with our configuration like this:

webpack.config.js

```
...

// Detect how npm is run and branch based on that
switch(process.env.npm_lifecycle_event) {
  case 'build':
    config = merge(
      common,
      {
        devtool: 'source-map'
      },
      parts.minify(),
      parts.setupCSS(PATHS.style)
    );
    break;
  default:
    ...
}

module.exports = validate(config);
}
```

If you execute `npm run build` now, you should see better results:

```
[webpack-validator] Config is valid.
Hash: aec016ce2e9d0dfa1577
Version: webpack 1.13.0
Time: 3342ms
     Asset       Size  Chunks             Chunk Names
    app.js      38 kB       0  [emitted]  app
app.js.map     325 kB       0  [emitted]  app
index.html  157 bytes          [emitted]
   [0] ./app/index.js 123 bytes {0} [built]
  [37] ./app/component.js 136 bytes {0} [built]
    + 36 hidden modules
Child html-webpack-plugin for "index.html":
        + 3 hidden modules
```

Given it needs to do more work, it took longer. But on the plus side the build is significantly smaller now.

Uglify warnings can help you to understand how it processes the code. Therefore it may be beneficial to have a peek at the full output every once in a while.

6.3 Controlling UglifyJS through Webpack

An UglifyJS feature known as **mangling** will be enabled by default. The feature will reduce local function and variable names to a minimum, usually to a single character. It can also rewrite properties to a more compact format if configured specifically.

Given these transformations can break your code, you have to be a little careful. A good example of this is Angular 1 and its dependency injection system. As it relies on strings, you must be careful not to mangle those or else it will fail to work.

Beyond mangling, it is possible to control all other UglifyJS features[1] through Webpack as illustrated below:

```
new webpack.optimize.UglifyJsPlugin({
  // Don't beautify output (enable for neater output)
  beautify: false,

  // Eliminate comments
  comments: false,

  // Compression specific options
  compress: {
    warnings: false,

    // Drop `console` statements
    drop_console: true
  },

  // Mangling specific options
  mangle: {
    // Don't mangle $
    except: ['$'],

    // Don't care about IE8
```

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

```
    screw_ie8 : true,

    // Don't mangle function names
    keep_fnames: true
  }
})
```

If you enable mangling, it is a good idea to set `except: ['webpackJsonp']` to avoid mangling the Webpack runtime.

Dropping the `console` statements can be achieved through Babel too by using the babel-plugin-remove-console[2] plugin. Babel is discussed in greater detail at the *Configuring React* chapter.

Yet another way to control Uglify would be to use the uglify-loader[3]. That gives yet another way to control minification behavior.

6.4 Conclusion

Even though our build is a little better now, there's still a fair amount of work left. The next simple step is to set an environment variable during the build to allow React to optimize itself. This technique can be used in your own code as well. You might want to skip certain checks in production usage and so on to bring the build size down.

[2] https://www.npmjs.com/package/babel-plugin-remove-console
[3] https://www.npmjs.com/package/uglify-loader

7. Setting Environment Variables

React relies on `process.env.NODE_ENV` based optimizations. If we force it to `production`, React will get built in an optimized manner. This will disable some checks (e.g., property type checks). Most importantly it will give you a smaller build and improved performance.

7.1 The Basic Idea of `DefinePlugin`

Webpack provides `DefinePlugin`. It is able to rewrite matching **free variables**. To understand the idea better, consider the example below:

```
var foo;

// Not free, not ok to replace
if(foo === 'bar') {
  console.log('bar');
}

// Free, ok to replace
if(bar === 'bar') {
  console.log('bar');
}
```

If we replaced `bar` with a string like `'bar'`, then we would end up with code like this:

```
var foo;

// Not free, not ok to replace
if(foo === 'bar') {
  console.log('bar');
}

// Free, ok to replace
if('bar' === 'bar') {
  console.log('bar');
}
```

Further analysis shows that `'bar' === 'bar'` equals `true` so UglifyJS gives us:

```
var foo;

// Not free, not ok to replace
if(foo === 'bar') {
  console.log('bar');
}

// Free, ok to replace
if(true) {
  console.log('bar');
}
```

And based on this UglifyJS can eliminate the `if` statement:

```
var foo;

// Not free, not ok to replace
if(foo === 'bar') {
  console.log('bar');
}

// Free, ok to replace
console.log('bar');
```

This is the core idea of `DefinePlugin`. We can toggle parts of code using it using this kind of mechanism. UglifyJS is able to perform the analysis for us and enable/disable entire portions of it as we prefer.

7.2 Setting `process.env.NODE_ENV`

To show you the idea in practice, we could have a declaration like `if(process.env.NODE_ENV === 'development')` within our code. Using `DefinePlugin` we could replace `process.env.NODE_ENV` with `'development'` to make our statement evaluate as true just like above.

As before, we can encapsulate this idea to a function:

libs/parts.js

```
...

exports.setFreeVariable = function(key, value) {
  const env = {};
  env[key] = JSON.stringify(value);

  return {
    plugins: [
      new webpack.DefinePlugin(env)
    ]
  };
}
```

We can connect this with our configuration like this:

webpack.config.js

```
...

// Detect how npm is run and branch based on that
switch(process.env.npm_lifecycle_event) {
  case 'build':
    config = merge(
      common,
      {
        devtool: 'source-map'
      },
      parts.setFreeVariable(
        'process.env.NODE_ENV',
        'production'
      ),
      parts.minify(),
      parts.setupCSS(PATHS.style)
    );
    break;
  default:
    ...
}

module.exports = validate(config);
```

Execute `npm run build` again, and you should see improved results:

```
[webpack-validator] Config is valid.
Hash: 9880a5782dc874c824c4
Version: webpack 1.13.0
Time: 3004ms
     Asset       Size  Chunks             Chunk Names
    app.js    25.4 kB       0  [emitted]  app
app.js.map     307 kB       0  [emitted]  app
index.html  157 bytes          [emitted]
   [0] ./app/index.js 123 bytes {0} [built]
  [36] ./app/component.js 136 bytes {0} [built]
    + 35 hidden modules
Child html-webpack-plugin for "index.html":
        + 3 hidden modules
```

So we went from 133 kB to 38 kB, and finally, to 25.4 kB. The final build is a little faster than the previous one. As that 25.4 kB can be served gzipped, it is quite reasonable. gzipping will drop around another 40%. It is well supported by browsers.

babel-plugin-transform-inline-environment-variables[1] Babel plugin can be used to achieve the same effect. See the official documentation[2] for details.

Note that we are missing react-dom[3] from our build. In practice our React application would be significantly larger unless we are using a lighter version such as preact[4] or react-lite[5]. These libraries might be missing some features, but they are worth knowing about if you use React.

7.3 Conclusion

Even though simply setting `process.env.NODE_ENV` the right way can help a lot especially with React related code, we can do better. We can split `app` and `vendor` bundles and add hashes to their filenames to benefit from browser caching. After all, the data that you don't need to fetch loads the fastest.

[1] https://www.npmjs.com/package/babel-plugin-transform-inline-environment-variables
[2] https://babeljs.io/docs/plugins/transform-inline-environment-variables/
[3] https://www.npmjs.com/package/react-dom
[4] https://www.npmjs.com/package/preact
[5] https://www.npmjs.com/package/react-lite

8. Splitting Bundles

Currently the production version of our application is a single JavaScript file. This isn't ideal. If we change the application, the client has to download vendor dependencies as well. It would be better to download only the changed portion. If the vendor dependencies change, then the client should fetch only the vendor dependencies. The same goes for actual application code.

This technique is known as **bundle splitting**. We can push the vendor dependencies to a bundle of its own and benefit from client level caching. We can do this in a such way that the whole size of the application remains the same. Given there are more requests to perform, there's a slight overhead. But the benefit of caching makes up for this cost.

To give you a simple example, instead of having *app.js* (100 kB), we could end up with *app.js* (10 kB) and *vendor.js* (90 kB). Now changes made to the application are cheap for the clients that have already used the application earlier.

Caching comes with its own problems. One of those is cache invalidation. We'll discuss a potential approach related to that in the next chapter. But before that, let's split some bundles.

8.1 Setting Up a `vendor` Bundle

So far our project has only a single entry named as `app`. As you might remember, our configuration tells Webpack to traverse dependencies starting from the `app` entry directory and then to output the resulting bundle below our `build` directory using the entry name and `.js` extension.

To improve the situation, we can define a `vendor` entry containing React. This is done by matching the dependency name. It is possible to generate this information automatically as discussed at the end of this chapter, but I'll go with a static array here to illustrate the basic idea. Change the code like this:

```
...

const common = {
  // Entry accepts a path or an object of entries.
  // We'll be using the latter form given it's
  // convenient with more complex configurations.
  entry: {
    app: PATHS.app,
    vendor: ['react']
-   app: PATHS.app
```

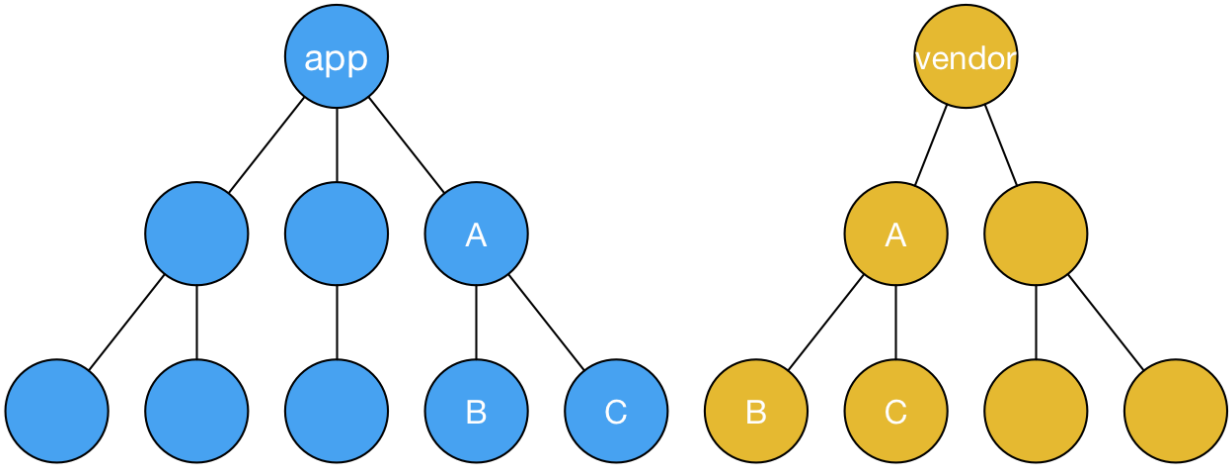
```
  },
  output: {
    path: PATHS.build,
    filename: '[name].js'
  },
  plugins: [
    new HtmlWebpackPlugin({
      title: 'Webpack demo'
    })
  ]
};

...
```

We have two separate entries, or **entry chunks**, now. The *Understanding Chunks* chapter digs into other available chunk types. Now we have a mapping between entries and the output configuration. `[name].js` will kick in based on the entry name and if you try to generate a build now (`npm run build`), you should see something like this:

```
[webpack-validator] Config is valid.
Hash: 6b55239dc87e2ae8efd6
Version: webpack 1.13.0
Time: 4168ms
        Asset       Size  Chunks             Chunk Names
       app.js    25.4 kB       0  [emitted]  app
    vendor.js    21.6 kB       1  [emitted]  vendor
   app.js.map     307 kB       0  [emitted]  app
vendor.js.map     277 kB       1  [emitted]  vendor
   index.html  190 bytes          [emitted]
   [0] ./app/index.js 123 bytes {0} [built]
   [0] multi vendor 28 bytes {1} [built]
  [36] ./app/component.js 136 bytes {0} [built]
    + 35 hidden modules
Child html-webpack-plugin for "index.html":
        + 3 hidden modules
```

app.js and *vendor.js* have separate chunk ids right now given they are entry chunks of their own. The output size is a little off, though. *app.js* should be significantly smaller to attain our goal with this build.

If you examine the resulting bundle, you can see that it contains React given that's how the default definition works. Webpack pulls the related dependencies to a bundle by default as illustrated by the image below:



Separate app and vendor bundles

A Webpack plugin known as `CommonsChunkPlugin` allows us alter this default behavior so that we can get the bundles we might expect.

8.2 Setting Up `CommonsChunkPlugin`

CommonsChunkPlugin[1] is a powerful and complex plugin. The use case we are covering here is a basic yet useful one. As before, we can define a function that wraps the basic idea.

To make our life easier in the future, we can make it extract a file known as a **manifest**. It contains the Webpack runtime that starts the whole application and contains the dependency information needed by it. This avoids a serious invalidation problem. Even though it's yet another file for the browser to load, it allows us to implement reliable caching in the next chapter.

If we don't extract a manifest, Webpack will generate the runtime to the vendor bundle. In case we modify the application code, the application bundle hash will change. Because that hash will change, so does the implementation of the runtime as it uses the hash to load the application bundle. Due to this the vendor bundle will receive a new hash too! This is why you should keep the manifest separate from the main bundles as doing this avoids the problem.

If you want to see this behavior in practice, try tweaking the implementation so that it **doesn't** generate the manifest after the next chapter. Change application code after that and see what happens to the generated code.

The following code combines the `entry` idea above with a basic `CommonsChunkPlugin` setup:

libs/parts.js

[1] https://webpack.github.io/docs/list-of-plugins.html#commonschunkplugin

```
...

exports.extractBundle = function(options) {
  const entry = {};
  entry[options.name] = options.entries;

  return {
    // Define an entry point needed for splitting.
    entry: entry,
    plugins: [
      // Extract bundle and manifest files. Manifest is
      // needed for reliable caching.
      new webpack.optimize.CommonsChunkPlugin({
        names: [options.name, 'manifest']
      })
    ]
  };
}
```

Given the function handles the entry for us, we can drop our `vendor` related configuration and use the function instead:

webpack.config.js

```
...

const common = {
  // Entry accepts a path or an object of entries.
  // We'll be using the latter form given it's
  // convenient with more complex configurations.
  entry: {
    app: PATHS.app
    app: PATHS.app,
    vendor: ['react']
  },
  output: {
    path: PATHS.build,
    filename: '[name].js'
  },
  plugins: [
    new HtmlWebpackPlugin({
      title: 'Webpack demo'
    })
```

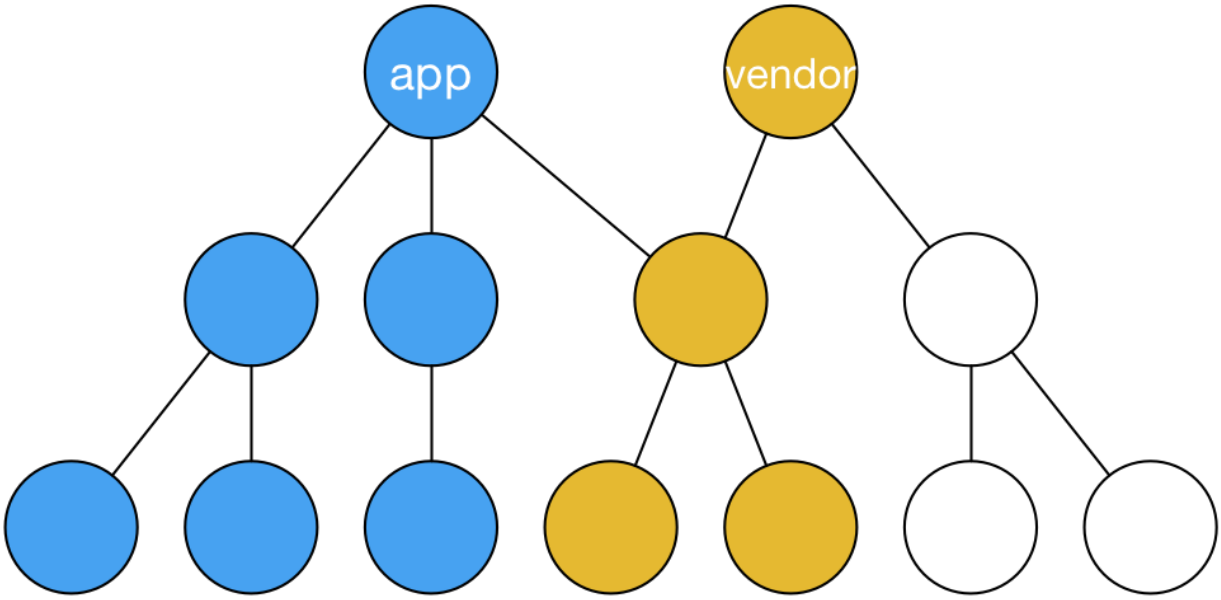
```
  ]
};

...

// Detect how npm is run and branch based on that
switch(process.env.npm_lifecycle_event) {
  case 'build':
    config = merge(
      common,
      {
        devtool: 'source-map'
      },
      parts.setFreeVariable(
        'process.env.NODE_ENV',
        'production'
      ),
      parts.extractBundle({
        name: 'vendor',
        entries: ['react']
      }),
      parts.minify(),
      parts.setupCSS(PATHS.style)
    );
    break;
  default:
    ...
}

module.exports = validate(config);
```

If you execute the build now using `npm run build`, you should see something along this:

```
[webpack-validator] Config is valid.
Hash: 516a574ca6ee19e87209
Version: webpack 1.13.0
Time: 2568ms
          Asset       Size  Chunks             Chunk Names
         app.js    3.94 kB    0, 2  [emitted]  app
      vendor.js    21.4 kB    1, 2  [emitted]  vendor
    manifest.js  780 bytes       2  [emitted]  manifest
     app.js.map    30.7 kB    0, 2  [emitted]  app
```

```
  vendor.js.map     274 kB    1, 2  [emitted]  vendor
manifest.js.map    8.72 kB       2  [emitted]  manifest
     index.html  225 bytes          [emitted]
   [0] ./app/index.js 123 bytes {0} [built]
   [0] multi vendor 28 bytes {1} [built]
  [36] ./app/component.js 136 bytes {0} [built]
    + 35 hidden modules
Child html-webpack-plugin for "index.html":
        + 3 hidden modules
```

Now our bundles look just the way we want. The image below illustrates the current situation:



App and vendor bundles after applying `CommonsChunkPlugin`

Beyond this, it is possible to define chunks that are loaded dynamically. This can be achieved through require.ensure[2]. We'll cover it in the *Understanding Chunks* chapter.

8.3 Loading `dependencies` to a `vendor` Bundle Automatically

If you maintain strict separation between `dependencies` and `devDependencies`, you can make Webpack to pick up your `vendor` dependencies automatically based on this information. You avoid having to manage those manually then. The basic idea goes like this:

[2] https://webpack.github.io/docs/code-splitting.html

```
...

const pkg = require('./package.json');

...

const common = {
  entry: {
    app: PATHS.app,
    vendor: Object.keys(pkg.dependencies)
  },
  ...
}

...
```

You can still exclude certain dependencies from the `vendor` entry point if you want by adding a bit of code for that. You can for instance `filter` out the dependencies you don't want there.

8.4 Conclusion

The situation is far better now. Note how small `app` bundle compared to the `vendor` bundle. In order to really benefit from this split, we should set up caching. This can be achieved by adding cache busting hashes to the filenames.

9. Adding Hashes to Filenames

Webpack relies on the concept of **placeholders**. These strings are used to attach specific information to Webpack output. The most useful ones are:

- `[path]` - Returns an entry path.
- `[name]` - Returns an entry name.
- `[hash]` - Returns the build hash.
- `[chunkhash]` - Returns a chunk specific hash.

Assuming we have configuration like this:

```
{
  output: {
    path: PATHS.build,
    filename: '[name].[chunkhash].js',
  }
}
```

We can generate filenames like these:

```
app.d587bbd6e38337f5accd.js
vendor.dc746a5db4ed650296e1.js
```

If the file contents related to a chunk are different, the hash will change as well, thus invalidating the cache. More accurately, the browser will send a new request for the new file. This means if only `app` bundle gets updated, only that file needs to be requested again.

An alternative way to achieve the same result would be to generate static filenames and invalidate the cache through a querystring (i.e., `app.js?d587bbd6e38337f5accd`). The part behind the question mark will invalidate the cache. This method is not recommended, though. According to Steve Souders[1], attaching the hash to the filename is the more performant way to go.

[1] http://www.stevesouders.com/blog/2008/08/23/revving-filenames-dont-use-querystring/

9.1 Setting Up Hashing

We have already split our application into `app.js` and `vendor.js` bundles and set up a separate `manifest` that bootstraps it. To get the hashing behavior we are after, we should generate `app.d587bbd6e38337f5accd.js` and `vendor.dc746a5db4ed650296e1.js` kind of files instead.

To make the setup work, our configuration is missing one vital part, the placeholders. Include them to the production configuration as follows:

webpack.config.js

```
...

// Detect how npm is run and branch based on that
switch(process.env.npm_lifecycle_event) {
  case 'build':
    config = merge(
      common,
      {
~~~~~~~~devtool: 'source-map'~~
        devtool: 'source-map',
        output: {
          path: PATHS.build,
          filename: '[name].[chunkhash].js',
          // This is used for require.ensure. The setup
          // will work without but this is useful to set.
          chunkFilename: '[chunkhash].js'
        }
      },
      ...
    );
    break;
  default:
    ...
}

module.exports = validate(config);
```

If you execute `npm run build` now, you should see output like this.

```
[webpack-validator] Config is valid.
Hash: 77395b0652b78e910b14
Version: webpack 1.13.0
Time: 2679ms
                              Asset       Size  Chunks             Chunk Names
         app.81e040e8c3dcc71d5624.js    3.96 kB    0, 2  [emitted]  app
       vendor.21dc91b20c0b1e6e16a1.js    21.4 kB    1, 2  [emitted]  vendor
    manifest.9e0e3d4035bea9b56f55.js  821 bytes       2  [emitted]  manifest
     app.81e040e8c3dcc71d5624.js.map    30.8 kB    0, 2  [emitted]  app
  vendor.21dc91b20c0b1e6e16a1.js.map     274 kB    1, 2  [emitted]  vendor
manifest.9e0e3d4035bea9b56f55.js.map    8.78 kB       2  [emitted]  manifest
                         index.html  288 bytes          [emitted]
   [0] ./app/index.js 123 bytes {0} [built]
   [0] multi vendor 28 bytes {1} [built]
  [36] ./app/component.js 136 bytes {0} [built]
    + 35 hidden modules
Child html-webpack-plugin for "index.html":
        + 3 hidden modules
```

Our files have neat hashes now. To prove that it works, you could try altering *app/index.js* and include a `console.log` there. After you build, only `app` and `manifest` related bundles should change.

One more way to improve the build further would be to load popular dependencies, such as React, through a CDN. That would decrease the size of the vendor bundle even further while adding an external dependency on the project. The idea is that if the user has hit the CDN earlier, caching can kick in just like here.

9.2 Conclusion

Even though our project has neat caching behavior now, adding hashes to our filenames brings a new problem. If a hash changes, we still have possible older files within our output directory. To eliminate this problem, we can set up a little plugin to clean it up for us.

10. Cleaning the Build

Our current setup doesn't clean the `build` directory between builds. As this can get annoying if we change our setup, we can use a plugin to clean the directory for us.

Another valid way to resolve the issue would be to handle this outside of Webpack. You could solve it on the system level through a npm script. In this case you would trigger `rm -rf ./build && webpack`. A task runner could work as well.

10.1 Setting Up *clean-webpack-plugin*

Install the clean-webpack-plugin[1] first:

```
npm i clean-webpack-plugin --save-dev
```

Next we need to define a little function to wrap the basic idea. We could use the plugin directly, but this feels like something that could be useful across projects so it makes sense to push to our library:

libs/parts.js

```
const webpack = require('webpack');
const CleanWebpackPlugin = require('clean-webpack-plugin');

...

exports.clean = function(path) {
  return {
    plugins: [
      new CleanWebpackPlugin([path], {
        // Without `root` CleanWebpackPlugin won't point to our
        // project and will fail to work.
        root: process.cwd()
      })
    ]
  };
}
```

We can connect it with our project like this:

webpack.config.js

[1] https://www.npmjs.com/package/clean-webpack-plugin

```
...

// Detect how npm is run and branch based on that
switch(process.env.npm_lifecycle_event) {
  case 'build':
    config = merge(
      common,
      {
        devtool: 'source-map',
        output: {
          path: PATHS.build,
          filename: '[name].[chunkhash].js',
          // This is used for require.ensure. The setup
          // will work without but this is useful to set.
          chunkFilename: '[chunkhash].js'
        }
      },
      parts.clean(PATHS.build),
      ...
    );
    break;
  default:
    ...
}

module.exports = validate(config);
```

After this change, our `build` directory should remain nice and tidy when building. You can verify this by building the project and making sure no old files remained in the output directory.

If you want to preserve possible dotfiles within your build directory, you can use `path.join(PATHS.build, '*')` instead of `PATHS.build`.

10.2 Conclusion

Our build is starting to get pretty neat now. There's one major issue, though. Our CSS has been inlined with JavaScript. This can result in the dreaded **Flash of Unstyled Content** (FOUC). It's also not ideal caching-wise.

A small change to the CSS would invalidate our `app` bundle. This doesn't hurt during development, but it's not something we want to experience in production. We can resolve this problem by separating our CSS to a file of its own.

11. Separating CSS

Even though we have a nice build set up now, where did all the CSS go? As per our configuration, it has been inlined to JavaScript! Even though this can be convenient during development, it doesn't sound ideal. The current solution doesn't allow us to cache CSS. In some cases we might suffer from a Flash of Unstyled Content (FOUC).

It just so happens that Webpack provides a means to generate a separate CSS bundle. We can achieve this using the ExtractTextPlugin[1]. It comes with overhead during the compilation phase, and it won't work with Hot Module Replacement (HMR) by design. Given we are using it only for production, that won't be a problem.

This same technique can be used with other assets, like templates, too.

It can be potentially dangerous to use inline styles in production as it represents an attack vector! Favor `ExtractTextPlugin` and similar solutions in production usage.

11.1 Setting Up *extract-text-webpack-plugin*

It will take some configuration to make it work. Install the plugin:

```
npm i extract-text-webpack-plugin --save-dev
```

The plugin operates in two parts. There's a loader, `ExtractTextPlugin.extract`, that marks the assets to be extracted. The plugin will perform its work based on this annotation. The idea looks like this:

libs/parts.js

[1] https://www.npmjs.com/package/extract-text-webpack-plugin

```
const webpack = require('webpack');
const CleanWebpackPlugin = require('clean-webpack-plugin');
const ExtractTextPlugin = require('extract-text-webpack-plugin');

...

exports.extractCSS = function(paths) {
  return {
    module: {
      loaders: [
        // Extract CSS during build
        {
          test: /\.css$/,
          loader: ExtractTextPlugin.extract('style', 'css'),
          include: paths
        }
      ]
    },
    plugins: [
      // Output extracted CSS to a file
      new ExtractTextPlugin('[name].[chunkhash].css')
    ]
  };
}
```

Connect the function with our configuration:

webpack.config.js

```
...

// Detect how npm is run and branch based on that
switch(process.env.npm_lifecycle_event) {
  case 'build':
    config = merge(
      ...
      parts.minify(),
      parts.extractCSS(PATHS.app)
      parts.setupCSS(PATHS.app)
    );
    break;
  default:
    config = merge(
```

```
      ...
    );
}

module.exports = validate(config);
```

Using this setup, we can still benefit from the HMR during development. For a production build, we generate a separate CSS, though. *html-webpack-plugin* will pick it up automatically and inject it into our `index.html`.

Definitions, such as `loaders: [ExtractTextPlugin.extract('style', 'css')]`, won't work and will cause the build to error instead! So when using `ExtractTextPlugin`, use the `loader` form instead.

If you want to pass more loaders to the `ExtractTextPlugin`, you should use `!` syntax. Example: `ExtractTextPlugin.extract('style', 'css!postcss')`.

After running `npm run build`, you should see output similar to the following:

```
[webpack-validator] Config is valid.
clean-webpack-plugin: .../webpack-demo/build has been removed.
Hash: 27832e316f572a80ce4f
Version: webpack 1.13.0
Time: 3084ms
                              Asset       Size  Chunks             Chunk Names
         app.c3162186fdfffbe6bbed.js  277 bytes    0, 2  [emitted]  app
       vendor.21dc91b20c0b1e6e16a1.js    21.4 kB    1, 2  [emitted]  vendor
    manifest.149335ad7c6634496b11.js  821 bytes       2  [emitted]  manifest
        app.c3162186fdfffbe6bbed.css   80 bytes    0, 2  [emitted]  app
     app.c3162186fdfffbe6bbed.js.map    1.77 kB    0, 2  [emitted]  app
    app.c3162186fdfffbe6bbed.css.map  105 bytes    0, 2  [emitted]  app
  vendor.21dc91b20c0b1e6e16a1.js.map     274 kB    1, 2  [emitted]  vendor
manifest.149335ad7c6634496b11.js.map    8.78 kB       2  [emitted]  manifest
                         index.html  347 bytes          [emitted]
   [0] ./app/index.js 123 bytes {0} [built]
   [0] multi vendor 28 bytes {1} [built]
  [36] ./app/component.js 136 bytes {0} [built]
    + 35 hidden modules
Child html-webpack-plugin for "index.html":
        + 3 hidden modules
Child extract-text-webpack-plugin:
        + 2 hidden modules
```

If you are getting `Module build failed: CssSyntaxError:` error, make sure your `common` configuration doesn't have a CSS related section set up!

Now our styling has been pushed to a separate CSS file. As a result, our JavaScript bundles have become slightly smaller. We also avoid the FOUC problem. The browser doesn't have to wait for JavaScript to load to get styling information. Instead, it can process the CSS separately avoiding the flash.

The current setup is fairly nice. There's one problem, though. If you try to modify either *index.js* or *main.css*, the hash of both files (*app.js* and *app.css*) will change! This is because they belong to the same entry chunk due to that `require` at *app/index.js*. The problem can be avoided by separating chunks further.

If you have a complex project with a lot of dependencies, it is likely a good idea to use the `DedupePlugin`. It will find possible duplicate files and deduplicate them. Use `new webpack.optimize.DedupePlugin()` in your plugins definition to enable it. You should use it in your production build.

11.2 Separating Application Code and Styling

A logical way to solve our chunk issue is to push application code and styling to separate entry chunks. This breaks the dependency and fixes caching. To achieve this we need to decouple styling from its current chunk and define a custom chunk for it through configuration:

app/index.js

```
require('react');

require('./main.css');

...
```

In addition, we need to define a separate entry for styling:

webpack.config.js

```
...

const PATHS = {
  app: path.join(__dirname, 'app'),
  style: path.join(__dirname, 'app', 'main.css'),
  build: path.join(__dirname, 'build')
};

const common = {
  // Entry accepts a path or an object of entries.
  // We'll be using the latter form given it's
  // convenient with more complex configurations.
  entry: {
    style: PATHS.style,
    app: PATHS.app
  },
  ...
};

// Detect how npm is run and branch based on that
switch(process.env.npm_lifecycle_event) {
  case 'build':
    config = merge(
      ...
      parts.minify(),
      parts.extractCSS(PATHS.style)
      parts.extractCSS(PATHS.app)
    );
    break;
  default:
    config = merge(
      ...
      parts.setupCSS(PATHS.style),
      parts.setupCSS(PATHS.app),
      parts.devServer({
        // Customize host/port here if needed
        host: process.env.HOST,
        port: process.env.PORT
      })
    );
}
```

```
module.exports = validate(config);
```

If you build the project now through `npm run build`, you should see something like this:

```
[webpack-validator] Config is valid.
clean-webpack-plugin: .../webpack-demo/build has been removed.
Hash: e6e6cecdefbb54c610c1
Version: webpack 1.13.0
Time: 2788ms
                              Asset       Size  Chunks             Chunk Names
     app.a51c1a5cde933b81dc3e.js.map    1.57 kB    0, 3  [emitted]  app
         app.a51c1a5cde933b81dc3e.js  252 bytes    0, 3  [emitted]  app
       vendor.6947db44af2e47a304eb.js    21.4 kB    2, 3  [emitted]  vendor
    manifest.c2487fa71892504eb968.js  846 bytes       3  [emitted]  manifest
      style.e5eae09a78b3efd50e73.css   82 bytes    1, 3  [emitted]  style
       style.e5eae09a78b3efd50e73.js   93 bytes    1, 3  [emitted]  style
   style.e5eae09a78b3efd50e73.js.map  430 bytes    1, 3  [emitted]  style
  style.e5eae09a78b3efd50e73.css.map  107 bytes    1, 3  [emitted]  style
  vendor.6947db44af2e47a304eb.js.map     274 kB    2, 3  [emitted]  vendor
manifest.c2487fa71892504eb968.js.map    8.86 kB       3  [emitted]  manifest
                         index.html  402 bytes          [emitted]
   [0] ./app/index.js 100 bytes {0} [built]
   [0] multi vendor 28 bytes {2} [built]
  [32] ./app/component.js 136 bytes {0} [built]
    + 35 hidden modules
Child html-webpack-plugin for "index.html":
        + 3 hidden modules
Child extract-text-webpack-plugin:
        + 2 hidden modules
```

After this step we have managed to separate styling from JavaScript. Changes made to it shouldn't affect JavaScript chunk hashes or vice versa. The approach comes with a small glitch, though.

If you look closely, you can see a file named *style.e5eae09a78b3efd50e73.js* in the output. Yours might be different. It is a file generated by Webpack and it looks like this:

```
webpackJsonp([1,3],[function(n,c){}]);
```

Technically it's redundant. It would be safe to exclude the file through a check at *HtmlWebpackPlugin* template. But this solution is good enough for the project. Ideally Webpack shouldn't generate these files at all and there's an issue[2] related to it.

[2] https://github.com/webpack/webpack/issues/1967

In the future we might be able to avoid this problem by using `[contenthash]` placeholder. It's generated based on file content (i.e., CSS in this case). Unfortunately it doesn't work as expected when the file is included in a chunk as in our original setup. This issue has been reported as Webpack issue #672[3].

11.3 Conclusion

Our current setup separates styling from JavaScript neatly. There is more we can do with CSS, though. Often big CSS frameworks come with plenty of CSS rules and a lot of those end up being unused. In the next chapter I will show you how to eliminate these rules from your build.

[3] https://github.com/webpack/webpack/issues/672

12. Eliminating Unused CSS

Frameworks like Bootstrap[1] tend to come with a lot of CSS. Often you use only a small part of it. Normally you just bundle even the unused CSS. It is possible, however, to eliminate the portions you aren't using. A tool known as purifycss[2] can achieve this by analyzing our files. It also works with single page applications.

12.1 Setting Up purifycss

Using purifycss can lead to great savings. In their example they purify and minify Bootstrap (140 kB) in an application using ~40% of its selectors to mere ~35 kB. That's a big difference.

Webpack plugin known as purifycss-webpack-plugin[3] allows us to achieve results like this. It is preferable to use the `ExtractTextPlugin` with it. Install it first:

```
npm i purifycss-webpack-plugin --save-dev
```

To make our demo more realistic, let's install a little CSS framework known as Pure.css[4] as well and refer to it from our project so that we can see purifycss in action:

```
npm i purecss --save-dev
```

We also need to refer to it from our configuration:

webpack.config.js

[1] https://getbootstrap.com/
[2] https://github.com/purifycss/purifycss
[3] https://www.npmjs.com/package/purifycss-webpack-plugin
[4] http://purecss.io/

```
...

const PATHS = {
  app: path.join(__dirname, 'app'),
  style: [
    path.join(__dirname, 'node_modules', 'purecss'),
    path.join(__dirname, 'app', 'main.css')
  ],
  style: path.join(__dirname, 'app', 'main.css'),
  build: path.join(__dirname, 'build')
};

...
```

Thanks to our path setup we don't need to tweak the remainder of the code. If you execute `npm run build`, you should see something like this:

```
[webpack-validator] Config is valid.
clean-webpack-plugin: .../webpack-demo/build has been removed.
Hash: adc32c7f82a388002a6e
Version: webpack 1.13.0
Time: 3656ms
                              Asset       Size  Chunks             Chunk Names
     app.a51c1a5cde933b81dc3e.js.map    1.57 kB    0, 3  [emitted]  app
         app.a51c1a5cde933b81dc3e.js  252 bytes    0, 3  [emitted]  app
      vendor.6947db44af2e47a304eb.js    21.4 kB    2, 3  [emitted]  vendor
    manifest.86e8bb3f3a596746a1a6.js  846 bytes       3  [emitted]  manifest
      style.e6624bc802ded7753823.css    16.7 kB    1, 3  [emitted]  style
       style.e6624bc802ded7753823.js  156 bytes    1, 3  [emitted]  style
   style.e6624bc802ded7753823.js.map  834 bytes    1, 3  [emitted]  style
  style.e6624bc802ded7753823.css.map  107 bytes    1, 3  [emitted]  style
  vendor.6947db44af2e47a304eb.js.map     274 kB    2, 3  [emitted]  vendor
manifest.86e8bb3f3a596746a1a6.js.map    8.86 kB       3  [emitted]  manifest
                         index.html  402 bytes          [emitted]
   [0] ./app/index.js 100 bytes {0} [built]
   [0] multi vendor 28 bytes {2} [built]
   [0] multi style 40 bytes {1} [built]
  [32] ./app/component.js 136 bytes {0} [built]
    + 37 hidden modules
Child html-webpack-plugin for "index.html":
        + 3 hidden modules
Child extract-text-webpack-plugin:
```

```
        + 2 hidden modules
Child extract-text-webpack-plugin:
        + 2 hidden modules
```

As you can see, `style.e6624bc802ded7753823.css` grew from 82 bytes to 16.7 kB as it should have. Also the hash changed because the file contents changed as well.

In order to give purifycss a chance to work and not eliminate whole PureCSS, we'll need to refer to it from our code. Add a `className` to our demo component like this:

app/component.js

```
module.exports = function () {
  var element = document.createElement('h1');

  element.className = 'pure-button';
  element.innerHTML = 'Hello world';

  return element;
};
```

If you run the application (`npm start`), our "Hello world" should look like a button.

We need one more bit, the configuration needed to make purifycss work. Expand parts like this:

libs/parts.js

```
const webpack = require('webpack');
const CleanWebpackPlugin = require('clean-webpack-plugin');
const ExtractTextPlugin = require('extract-text-webpack-plugin');
const PurifyCSSPlugin = require('purifycss-webpack-plugin');

...

exports.purifyCSS = function(paths) {
  return {
    plugins: [
      new PurifyCSSPlugin({
        basePath: process.cwd(),
        // `paths` is used to point PurifyCSS to files not
        // visible to Webpack. You can pass glob patterns
        // to it.
        paths: paths
      }),
```

```
    ]
  }
}
```

Next we need to connect this part to our configuration. It is important the plugin is used *after* the `ExtractTextPlugin` as otherwise it won't work!

webpack.config.js

```
...

// Detect how npm is run and branch based on that
switch(process.env.npm_lifecycle_event) {
  case 'build':
    config = merge(
      ...
      parts.minify(),
      parts.extractCSS(PATHS.style),
      parts.purifyCSS([PATHS.app])
      parts.extractCSS(PATHS.style)
    );
  default:
    ...

module.exports = validate(config);
```

Given Webpack is aware of `PATHS.app` through an entry, we could skip passing it to `parts.purifyCSS`. As explicit is often nicer than implicit, having it here doesn't hurt. We'll get the same result either way.

If you execute `npm run build` now, you should see something like this:

```
[webpack-validator] Config is valid.
clean-webpack-plugin: .../webpack-demo/build has been removed.
Hash: 7eaf3b6bae4156774447
Version: webpack 1.13.0
Time: 8703ms
                             Asset       Size  Chunks             Chunk Names
     app.a26b058bec8ce4d237ff.js.map    1.57 kB    0, 3  [emitted]  app
         app.a26b058bec8ce4d237ff.js  252 bytes    0, 3  [emitted]  app
      vendor.6947db44af2e47a304eb.js    21.4 kB    2, 3  [emitted]  vendor
    manifest.79745ac6c18fa88e9d61.js  846 bytes       3  [emitted]  manifest
      style.e6624bc802ded7753823.css    13.1 kB    1, 3  [emitted]  style
```

```
       style.e6624bc802ded7753823.js  156 bytes    1, 3  [emitted]  style
    style.e6624bc802ded7753823.js.map  834 bytes    1, 3  [emitted]  style
   style.e6624bc802ded7753823.css.map  107 bytes    1, 3  [emitted]  style
   vendor.6947db44af2e47a304eb.js.map     274 kB    2, 3  [emitted]  vendor
manifest.79745ac6c18fa88e9d61.js.map    8.86 kB       3  [emitted]  manifest
                         index.html  402 bytes          [emitted]
   [0] ./app/index.js 100 bytes {0} [built]
   [0] multi vendor 28 bytes {2} [built]
   [0] multi style 40 bytes {1} [built]
  [32] ./app/component.js 137 bytes {0} [built]
    + 37 hidden modules
Child html-webpack-plugin for "index.html":
        + 3 hidden modules
Child extract-text-webpack-plugin:
        + 2 hidden modules
Child extract-text-webpack-plugin:
        + 2 hidden modules
```

The size of our style went from 16.7 kB to 13.1 kB. It is not a huge difference in this case, but it is still something. It is interesting to note that processing time went from three seconds to eight so there is a cost involved! The technique is useful to know as it will likely come in handy with heavier CSS frameworks.

PurifyCSS supports additional options[5]. You could for example enable additional logging by setting `purifyOptions: {info: true}` when instantiating the plugin.

12.2 Conclusion

Build-wise our little project is starting to get there. Now our CSS is separate and pure. In the next chapter I'll show you how to analyze Webpack build statistics so you understand better what the generated bundles actually contain.

[5] https://github.com/purifycss/purifycss#the-optional-options-argument

13. Analyzing Build Statistics

Analyzing build statistics is a good step towards understanding Webpack better. We can get statistics from it easily and we can visualize them using a tool. This shows us the composition of our bundles.

13.1 Configuring Webpack

In order to get suitable output we'll need to do a couple of tweaks to our configuration. We'll need to enable two flags:

- `--profile` to capture timing related information.
- `--json` to make Webpack output those statistics we want.

Here's the line of code we need to pipe the output to a file:

package.json

```
{
  ...
  "scripts": {
    "stats": "webpack --profile --json > stats.json",
    ...
  },
  ...
}
```

This is the basic setup you'll need regardless of your Webpack configuration.

To adapt the tutorial configuration to work with the new npm script, we'll want to make sure it evaluates the build output. That gives realistic results to us. Adjust the configuration as follows:

webpack.config.js

```
...


// Detect how npm is run and branch based on that
switch(process.env.npm_lifecycle_event) {
  case 'build':
  case 'stats':
    config = merge(
      ...
    );
    break;
  default:
    config = merge(
      ...
    );
}

~~module.exports = validate(config);~~
// Run validator in quiet mode to avoid output in stats
module.exports = validate(config, {
  quiet: true
});
```

If you execute `npm run stats` now, you should find *stats.json* at your project root after it has finished processing. Even though having a look at the file itself gives you some idea of what's going on, often it's preferable to use a specific tool for that. I've listed a few alternatives below:

- The official analyse tool[1] gives you recommendations and a good idea of your application dependency graph. Source[2].
- Webpack Visualizer[3] provides a pie chart showing your bundle composition. This is handy for understanding which dependencies contribute to the size of the overall result.
- Webpack Chart[4] is another similar visualization.
- robertknight/webpack-bundle-size-analyzer[5] gives a text based composition.

When you are optimizing the size of your bundle output, these tools are invaluable. The official tool has the most functionality, but even a simple visualization can reveal problem spots.

[1] http://webpack.github.io/analyse/
[2] https://github.com/webpack/analyse
[3] https://chrisbateman.github.io/webpack-visualizer/
[4] https://alexkuz.github.io/webpack-chart/
[5] https://github.com/robertknight/webpack-bundle-size-analyzer

13.2 Webpack Stats Plugin

If you want to manage stats through a plugin, check out stats-webpack-plugin[6]. It gives you a bit more control over the output. You can use it to exclude certain dependencies from the output for example.

13.3 Conclusion

It is useful to analyze your build output. You can use the same technique with older projects to understand their composition.

To complete our setup, we'll set up a little deployment script that will allow us to push build output to GitHub Pages. In practice you would use something more sophisticated, but it's enough to illustrate the idea.

[6] https://www.npmjs.com/package/stats-webpack-plugin

14. Hosting on GitHub Pages

A package known as gh-pages[1] allows us host our application on GitHub easily. You point it to your build directory first. It will then pick up the contents and push them to the `gh-pages` branch.

14.1 Setting Up *gh-pages*

To get started, execute

```
npm i gh-pages --save-dev
```

We are also going to need an entry point at *package.json*:

package.json

```
{
  ...
  "scripts": {
    "deploy": "gh-pages -d build",
    ...
  },
  ...
}
```

To make the asset paths work on GitHub Pages, we also need to tweak a Webpack setting known as `output.publicPath`. It gives us control over the resulting urls you see at *index.html* for instance. If you are hosting your assets on a CDN, this would be the place to tweak. In this case it's enough to set it to point the GitHub project like this:

[1] https://www.npmjs.com/package/gh-pages

```
...

// Detect how npm is run and branch based on that
switch(process.env.npm_lifecycle_event) {
  case 'build':
  case 'stats':
    config = merge(
      common,
      {
        devtool: 'source-map',
        output: {
          path: PATHS.build,
          // Tweak this to match your GitHub project name
          publicPath: '/webpack-demo/'
          filename: '[name].[chunkhash].js',
          chunkFilename: '[chunkhash].js'
        }
      },
      ...
    );
    break;
  default:
    ...
}
```

If you execute `npm run deploy` now and everything goes fine, you should have your application hosted through GitHub Pages. You should find it at `https://<name>.github.io/<project>` (`github.com/<name>/<project>` at GitHub) assuming it worked.

If you need a more elaborate setup, you can use the Node.js API that *gh-pages* provides. The default CLI tool it provides is often enough, though.

14.2 Conclusion

The same idea works with other environments too. You can set up *gh-pages* to push into a branch you want. After this step we have a fairly complete development and production setup.

We'll discuss various Webpack related techniques in greater detail in the following parts of this book.

III Loading Assets

This part discusses how to load various common assets, such as styling files, images, and fonts, using Webpack.

It is possible to use many of these loaders within Babel environment using a Babel plugin known as babel-plugin-webpack-loaders[2]. This can help you to avoid a build step.

[2] https://www.npmjs.com/package/babel-plugin-webpack-loaders

15. Formats Supported

Webpack supports a large variety of formats through *loaders*. In addition, it supports a couple of JavaScript module formats out of the box. Generally, the idea is always the same. You always set up a loader, or loaders, and connect those with your directory structure. The system relies on configuration. Consider the example below where we set Webpack to load CSS:

webpack.config.js

```
...

module.exports = {
  ...
  module: {
    loaders: [
      {
        // Match files against RegExp
        test: /\.css$/,

        // Apply loaders against it. These need to
        // be installed separately. In this case our
        // project would need *style-loader* and *css-loader*.
        loaders: ['style', 'css'],

        // Restrict matching to a directory. This also accepts an array of paths.
        // Although optional, I prefer to set this (better performance,
        // clearer configuration).
        include: path.join(__dirname, 'app')
      }
    ]
  }
};
```

Webpack's loader definition is almost too flexible. I'll cover variants in the next chapter. Before that we can take a quick look at JavaScript module formats supported by Webpack.

If you are not sure how a particular RegExp matches, consider using an online tool, such as regex101[1].

[1] https://regex101.com/

15.1 JavaScript Module Formats Supported by Webpack

Webpack allows you to use different module formats, but under the hood they all work the same way. Most importantly you get CommonJS and AMD support out of the box. Webpack 2 will support ES6 module definition as well. For now, you have to stick with Babel[2] and babel-loader[3] to attain ES6 support.

I'll give you brief examples of the modules supported next so you have a better idea of what they look like. I consider CommonJS and AMD legacy formats. If possible, stick to ES6. Due to the definition characteristics, it's not entirely comparable with CommonJS, but it's enough for most use cases.

CommonJS

If you have used Node.js, it is likely that you are familiar with CommonJS already. Here's a brief example:

```
var MyModule = require('./MyModule');

// export at module root
module.exports = function() { ... };

// alternatively, export individual functions
exports.hello = function() {...};
```

ES6

ES6 is the format we all have been waiting for since 1995. As you can see, it resembles CommonJS a little bit and is quite clear!

```
import MyModule from './MyModule.js';

// export at module root
export default function () { ... };

// or export as module function,
// you can have multiple of these per module
export function hello() {...};
```

[2] https://babeljs.io

[3] https://www.npmjs.org/package/babel-loader

Webpack doesn't support this format out of the box yet so you will have to use babel-loader[4]. Webpack 2 will change the situation.

AMD

AMD, or asynchronous module definition, was invented as a workaround and popularized by RequireJS[5] script loader. It introduced a `define` wrapper:

```
define(['./MyModule.js'], function (MyModule) {
  // export at module root
  return function() {};
});

// or
define(['./MyModule.js'], function (MyModule) {
  // export as module function
  return {
    hello: function() {...}
  };
});
```

Incidentally, it is possible to use `require` within the wrapper like this:

```
define(['require'], function (require) {
  var MyModule = require('./MyModule.js');

  return function() {...};
});
```

This latter approach definitely eliminates some of the clutter. You will still end up with some code that might feel redundant. Given there's ES6 now, it probably doesn't make much sense to use AMD anymore unless you really have to.

UMD

UMD, universal module definition, takes it all to the next level. It is a monster of a format that aims to make the aforementioned formats compatible with each other. I will spare your eyes from

[4] https://www.npmjs.com/package/babel-loader
[5] http://requirejs.org/

it. Never write it yourself, leave it to the tools. If that didn't scare you off, check out the official definitions[6].

Webpack can generate UMD wrappers for you (`output.libraryTarget: 'umd'`). This is particularly useful for package authors. We'll get back to this later in the *Authoring Packages* chapter.

15.2 Conclusion

Webpack supports a large variety of file formats. More often than not you will have to install some loader. Webpack itself supports just a couple of common JavaScript module formats.

I will discuss Webpack's loader definitions in detail next.

[6] https://github.com/umdjs/umd

16. Loader Definitions

Webpack provides multiple ways to set up module loaders. I'll cover the most important ones next. Generally you either use `loader` (accepts string) or `loaders` field (accepts array of strings) and then pass possible query parameters using one of the available methods.

I recommend maintaining an `include` definition per each loader definition. This will restrict its search path, improve performance, and make your configuration easier to follow. `include` accepts either a path or an array of paths.

It can be a good idea to prefer absolute paths here as it allows you to move configuration without breaking assumptions. Ideally you have to tweak just a single place during restructuring.

16.1 Loader Evaluation Order

It is good to keep in mind that Webpack's `loaders` are always evaluated from right to left and from bottom to top (separate definitions). The right to left rule is easier to remember when you think about it in terms of functions. You can read definition `loaders: ['style', 'css']` as `style(css(input))` based on this rule. The following examples are equivalent as well:

```
{
  test: /\.css$/,
  loaders: ['style', 'css'],
  include: PATHS.app
}


{
  test: /\.css$/,
  loaders: ['style'],
  include: PATHS.app
},
{
  test: /\.css$/,
  loaders: ['css'],
  include: PATHS.app
}
```

The `loaders` of the latter definition could be rewritten in the query format discussed above after performing a split like this.

16.2 Passing Parameters to a Loader

Sometimes you might want to pass query parameters to a loader. By default you could do it through a query string:

```
{
  test: /\.jsx?$/,
  loaders: [
    'babel?cacheDirectory,presets[]=react,presets[]=es2015'
  ],
  include: PATHS.app
}
```

The problem with this approach is that it isn't particularly readable. A better way is to use the combination of `loader` and `query` fields:

```
{
  test: /\.jsx?$/,
  loader: 'babel',
  query: {
    cacheDirectory: true,
    presets: ['react', 'es2015']
  },
  include: PATHS.app
}
```

This approach becomes problematic with multiple loaders since it's limited just to one loader at a time. If you want to use this format with multiple, you need separate declarations.

Another way to deal with query parameters would be to rely on Node.js querystring[1] module and stringify structures through it so they can be passed through a `loaders` definition.

16.3 Conclusion

Webpack provides multiple ways to set up loaders. You should be careful especially with loader ordering. Sometimes naming (i.e, `loader` vs. `loaders`) can cause mysterious issues as well.

I will discuss specific assets types and how to load them using Webpack next.

[1] https://nodejs.org/api/querystring.html

17. Loading Styles

Loading styles is a standard operation. There are a lot of variants depending on the styling approach you use, though. I'll cover the most common options next. You can combine these approaches with the `ExtractTextPlugin` to get better output for your production build.

17.1 Loading CSS

Loading vanilla CSS is fairly straightforward as you can see in the example below. It parses the styles in the given `include` path (accepts an array too) while making sure only files ending with `.css` are matched. The definition then applies both *style-loader* and *css-loader* on it:

webpack.config.js

```
const common = {
  ...
  module: {
    loaders: [
      {
        test: /\.css$/,
        loaders: ['style', 'css'],
        include: PATHS.style
      }
    ]
  },
  ...
};
```

When Webpack evaluates the files, first css-loader[1] goes through possible `@import` and `url()` statements within the matched files and treats them as regular `require`. This allows us to rely on various other loaders, such as file-loader[2] or url-loader[3]. We will see how these work in the next chapters.

file-loader generates files, whereas *url-loader* can create inline data URLs for small resources. This can be useful for optimizing application loading. You avoid unnecessary requests while providing a

[1] https://www.npmjs.com/package/css-loader
[2] https://www.npmjs.com/package/file-loader
[3] https://www.npmjs.com/package/url-loader

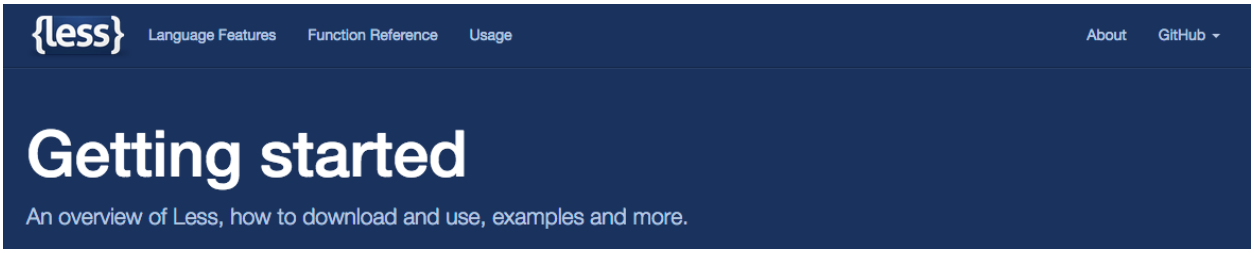
slightly bigger payload. Small improvements can yield large benefits if you depend on a lot of small resources in your style definitions.

After *css-loader* has done its part, *style-loader* picks up the output and injects the CSS into the resulting bundle. This will be inlined JavaScript by default. This is something you want to avoid in production usage. It makes sense to use `ExtractTextPlugin` to generate a separate CSS file in this case as we saw earlier.

Setting up other formats than vanilla CSS is simple as well. I'll discuss specific examples next.

If you want to enable sourcemaps for CSS, you should use `['style', 'css?sourceMap']` and set `output.publicPath` to an absolute url. *css-loader* issue 29[4] discusses this problem further.

17.2 Loading LESS



Less

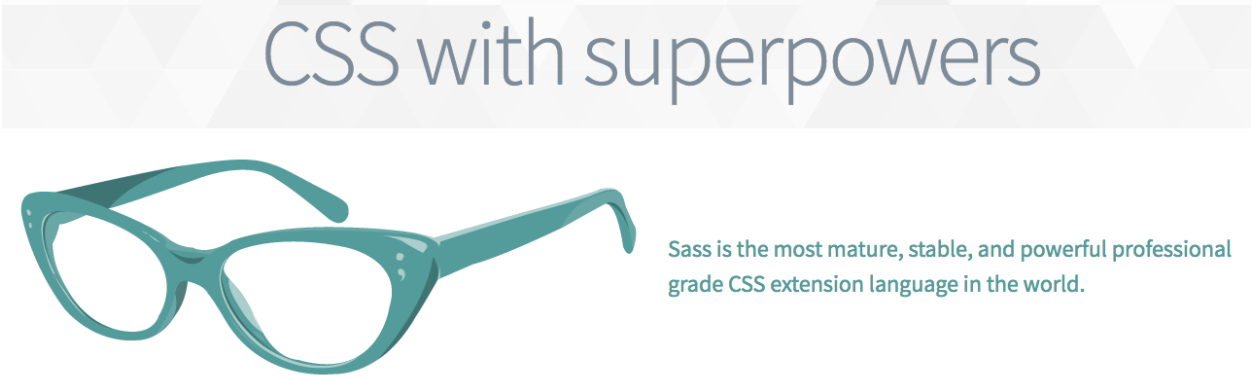
Less[5] is a popular CSS processor that is packed with functionality. In Webpack using Less doesn't take a lot of effort. less-loader[6] deals with the heavy lifting. You should install less[7] as well given it's a peer dependency of *less-loader*. Consider the following minimal setup:

```
{
  test: /\.less$/,
  loaders: ['style', 'css', 'less'],
  include: PATHS.style
}
```

There is also support for Less plugins, sourcemaps, and so on. To understand how those work you should check out the project itself.

[4] https://github.com/webpack/css-loader/issues/29
[5] http://lesscss.org/
[6] https://www.npmjs.com/package/less-loader
[7] https://www.npmjs.com/package/less

17.3 Loading SASS



Sass

Sass[8] is a popular alternative to Less. You should use sass-loader[9] with it. Remember to install node-sass[10] to your project as the loader has a peer dependency on that. Webpack doesn't take much configuration:

webpack.config.js

```
{
  test: /\.scss$/,
  loaders: ['style', 'css', 'sass'],
  include: PATHS.style
}
```

Check out the loader for more advanced usage.

Imports in LESS and SASS

If you import one LESS/SASS file from an other, use the exact same pattern as anywhere else. Webpack will dig into these files and figure out the dependencies.

```
@import "./variables.less";
```

You can also load LESS files directly from your node_modules directory. This is handy with libraries like Twitter Bootstrap:

[8] http://sass-lang.com/
[9] https://www.npmjs.com/package/sass-loader
[10] https://www.npmjs.com/package/node-sass

```
$import "~bootstrap/less/bootstrap";
```

17.4 Loading Stylus and YETICSS

Stylus

Stylus is yet another example of a CSS processor. It works well through stylus-loader[11]. There's also a pattern library known as yeticss[12] that works well with it. Consider the following configuration:

webpack.config.js

```
const common = {
  ...
  module: {
    loaders: [
      {
        test: /\.styl$/,
        loaders: ['style', 'css', 'stylus'],
        include: PATHS.style
      }
    ]
```

[11] https://github.com/shama/stylus-loader
[12] https://www.npmjs.com/package/yeticss

```
  },
  // yeticss
  stylus: {
    use: [require('yeticss')]
  }
};
```

To start using yeticss with Stylus, you must import it to one of your app's .styl files:

```
@import 'yeticss'

//or
@import 'yeticss/components/type'
```

17.5 PostCSS

PostCSS

PostCSS[13] allows you to perform transformations over CSS through JavaScript plugins. You can even find plugins that provide you Sass-like features. PostCSS can be thought as the equivalent of Babel for styling. It can be used through postcss-loader[14] with Webpack.

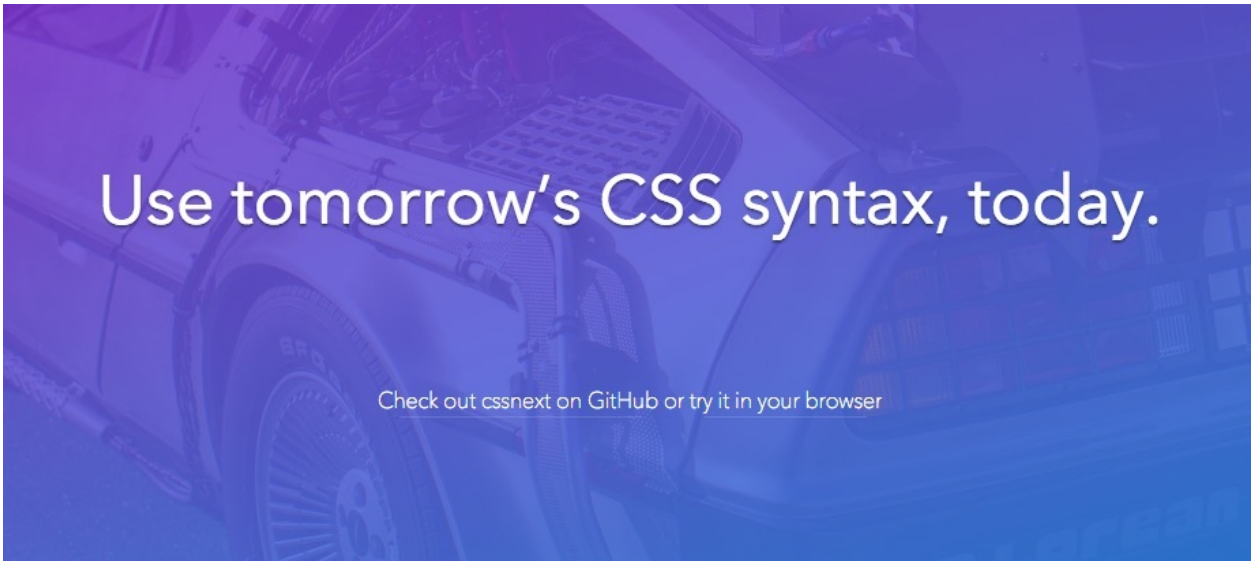
The example below illustrates how to set up autoprefixing using it. It also sets up precss[15], a PostCSS plugin that allows you to use Sass-like markup in your CSS. You can mix this technique with other loaders to allow autoprefixing there.

[13] https://github.com/postcss/postcss
[14] https://www.npmjs.com/package/postcss-loader
[15] https://www.npmjs.com/package/precss

webpack.config.js

```
const autoprefixer = require('autoprefixer');
const precss = require('precss');

module.exports = {
  module: {
    loaders: [
      {
        test: /\.css$/,
        loaders: ['style', 'css', 'postcss'],
        include: PATHS.style
      }
    ]
  },
  // PostCSS plugins go here
  postcss: function () {
      return [autoprefixer, precss];
  }
};
```

cssnext



cssnext

cssnext[16] is a PostCSS plugin that allows us to experience the future now. There are some restrictions,

[16] https://cssnext.github.io/

but it may be worth a go. In Webpack it is simply a matter of installing cssnext-loader[17] and attaching it to your CSS configuration. In our case, you would end up with the following:

webpack.config.js

```
{
  test: /\.css$/,
  loaders: ['style', 'css', 'cssnext'],
  include: PATHS.style
}
```

Alternatively, you could consume it through *postcss-loader* as a plugin if you need more control.

17.6 Conclusion

Loading style files through Webpack is fairly straight-forward. It supports even advanced techniques like CSS Modules[18]. CSS Modules make CSS local by default. This can be a great boon especially for developers who work with component oriented libraries. The approach works beautifully there.

[17] https://www.npmjs.com/package/cssnext-loader
[18] https://github.com/css-modules/webpack-demo

18. Loading Images

The easiest way to make your application slow is to load a lot of small assets. Each request comes with an overhead after all. HTTP/2 will help in this regard and change the situation somewhat drastically. Till then we are stuck with different approaches. Webpack allows a few of these. They are particularly relevant for loading images.

Webpack allows you to inline assets by using url-loader[1]. It will output your images as BASE64 strings within your JavaScript bundles. This will decrease the amount of requests needed while growing the bundle size. Given too large bundles aren't fun either, Webpack allows you to control the inlining process and defer loading to file-loader[2] that outputs image files and returns paths to them.

18.1 Setting Up *url-loader*

If you want to use *url-loader* and its *limit* feature, you will need to install both *url-loader* and *file-loader* to your project. Assuming you have configured your styles correctly, Webpack will resolve any `url()` statements your styling might have. You can of course point to the image assets through your JavaScript code as well.

In order to load *.jpg* and *.png* files while inlining files below 25kB, we would set up a loader like this:

```
{
  test: /\.(jpg|png)$/,
  loader: 'url?limit=25000',
  include: PATHS.images
}
```

18.2 Setting Up *file-loader*

If you want to skip inlining, you can use *file-loader* directly. The following setup customizes the resulting filename. By default *file-loader* returns the MD5 hash of the file's contents with the original extension:

[1] https://www.npmjs.com/package/url-loader
[2] https://www.npmjs.com/package/file-loader

```
{
  test: /\.(jpg|png)$/,
  loader: 'file?name=[path][name].[hash].[ext]',
  include: PATHS.images
}
```

18.3 Loading SVGs

Webpack has a few ways[3] to load SVGs. However the simplest way is through *file-loader* as follows:

```
{
  test: /\.svg$/,
  loader: 'file',
  include: PATHS.images
}
```

Assuming you have set up your styling correctly, you can refer to your SVG files like this. The example SVG path below is relative to the CSS file:

```
.icon {
   background-image: url('../assets/icon.svg');
}
```

If you want the raw SVG content, you can use the raw-loader[4] for this purpose. This can be useful if you want to inject the SVG content to directly to JavaScript or HTML markup.

18.4 Compressing Images

In case you want to compress your images, use image-webpack-loader[5] or svgo-loader[6] (SVG specific). This type of loader should be applied first to the data so remember to place it as the last within `loaders` listing.

Compression is particularly useful during production usage as it will decrease the amount of bandwidth required to download your image assets and speed up your site or application as a result.

[3] https://github.com/webpack/webpack/issues/595
[4] https://www.npmjs.com/package/raw-loader
[5] https://www.npmjs.com/package/image-webpack-loader
[6] https://github.com/pozadi/svgo-loader

18.5 Conclusion

Webpack allows you to inline images within your bundles when needed. Figuring out good inlining limits for your images might take some experimentation. You have to balance between bundle sizes and the amount of requests.

19. Loading Fonts

Loading fonts is a surprisingly tough problem. There are typically four(!) font formats to worry about, each for certain browser. Inlining all formats at once wouldn't be a particularly good idea. There are a couple of strategies we can consider.

19.1 Choosing One Format

Depending on your project requirements, you might be able to get away with less formats. If you exclude Opera Mini, all browsers support the *.woff* format. The render result may differ depending on the browser so you might want to experiment here.

If we go with just one format, we can use a similar setup as for images and rely on both *file-loader* and *url-loader* while using the limit option:

```
{
  test: /\.woff$/,
  loader: 'url?limit=50000',
  include: PATHS.fonts
}
```

A more elaborate way to achieve a similar result would be to use:

```
{
  // Match woff2 in addition to patterns like .woff?v=1.1.1.
  test: /\.woff(2)?(\?v=[0-9]\.[0-9]\.[0-9])?$/,
  loader: 'url',
  query: {
    limit: 50000,
    mimetype: 'application/font-woff',
    name: './fonts/[hash].[ext]'
  }
}
```

19.2 Supporting Multiple Formats

In case we want to make sure our site looks good on a maximum amount of browsers, we might as well use just *file-loader* and forget about inlining. Again, it's a trade-off as we get extra requests, but perhaps it's the right move. Here we could end up with a loader configuration like this:

```
{
  test: /\.woff$/,
  // Inline small woff files and output them below font/.
  // Set mimetype just in case.
  loader: 'url',
  query: {
    name: 'font/[hash].[ext]',
    limit: 5000,
    mimetype: 'application/font-woff'
  },
  include: PATHS.fonts
},
{
  test: /\.ttf$|\.eot$/,
  loader: 'file',
  query: {
    name: 'font/[hash].[ext]'
  },
  include: PATHS.fonts
}
```

19.3 Conclusion

Loading fonts is similar as loading other assets. Here we have extra concerns to consider. We need to consider the browsers we want to support and choose the loading strategy based on that.

IV Advanced Techniques

This part covers advanced techniques related to Webpack. First we'll discuss Webpack's chunks in detail. Understanding the topic gives you more flexibility as you can load assets as you need them. We'll also take a peek at package author specific Webpack options. We'll learn to implement basic loaders and configure React in particular.

20. Understanding Chunks

Chunks are one of the most fundamental concepts of Webpack. We already touched the topic at the *Splitting Bundles* chapter. There we set up two separate entries, one for the application code and one for our vendor dependencies.

With some additional setup Webpack was able to output separate bundles for these. We also did something similar with CSS as we separated it from our code to a bundle of its own in order to improve caching behavior.

20.1 Chunk Types

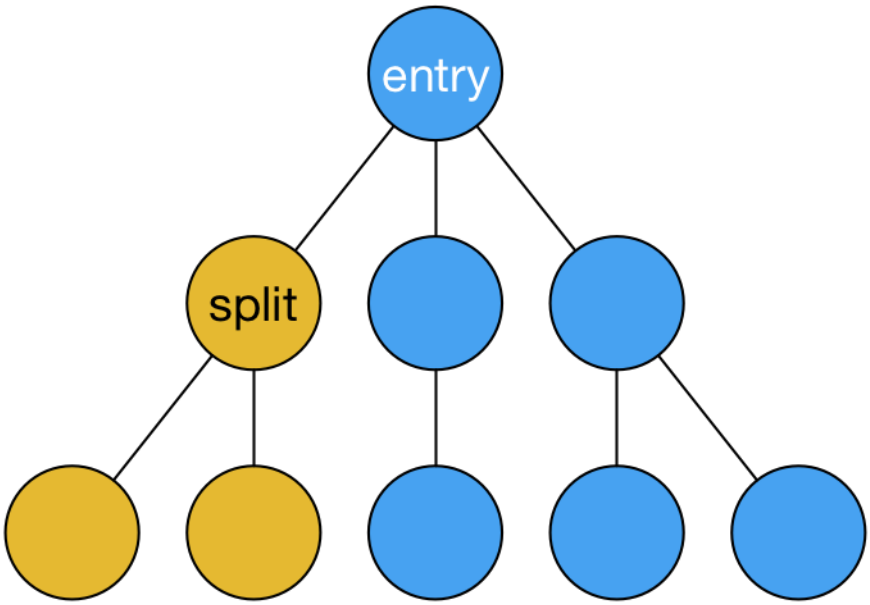
As discussed in the documentation[1], internally Webpack treats chunks in three types:

- **Entry chunks** - Entry chunks contain Webpack runtime and modules it then loads. So far we've been dealing with these.
- **Normal chunks** - Normal chunks **don't** contain Webpack runtime. Instead, these can be loaded dynamically while the application is running. A suitable wrapper (JSONP for example) is generated for these.
- **Initial chunks** - Initial chunks are normal chunks that count towards initial loading time of the application. These are generated by the `CommonsChunkPlugin`. As a user you don't have to care about these. It's the split between entry chunks and normal chunks that's important.

Given we've covered how to deal with entry chunks already, I won't delve into that. Instead, I'm going to show you how to deal with normal chunks. They enable one of the most powerful features of Webpack, lazy loading.

[1] https://webpack.github.io/docs/code-splitting.html#chunk-types

20.2 Lazy Loading with Webpack



Bundle with a dynamically loaded normal chunk

What makes Webpack powerful is its capability of splitting up your application into smaller chunks to load. This is far more flexible than standard concatenation. Even though concatenation works, it's not always a good approach. This is particularly true when the size of your application begins to grow.

Often you don't need all of the dependencies at once. As we saw earlier, you can split your dependencies and benefit from browser caching behavior. This is a good step, but it's not enough always. Your bundles can still be somewhat big. *Lazy loading* allows us to go further.

Introduction to Lazy Loaded Search with *lunr*

Let's say we want to implement a rough little search for our application without a proper search back-end. We might want to use something like lunr[2] for generating an index to search against.

The problem is that the index can be quite big depending on the amount of the content. The dumb way to implement this kind of search would be to include the index required to the application bundle itself and then perform search against that.

The good thing is that we don't actually need the search index straight from the start. We can do something more clever. We can start loading the index when the user selects our search field.

[2] http://lunrjs.com/

This defers the loading and moves it to a place where it's more acceptable. Given the initial search might be slower than the subsequent ones we could display a loading indicator. But that's fine from the user point of view.

Implementing Search with Lazy Loading

Implementing this idea is straight-forward. We need to capture when the user selects the search element, load the data unless it has been loaded already, and then execute our search logic against it. In React we could end up with something like this:

App.jsx

```
import React from 'react';

export default class App extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      index: null,
      value: '',
      lines: [],
      results: []
    };

    this.onChange = this.onChange.bind(this);
  }
  render() {
    const results = this.state.results;

    return (
      <div className="app-container">
        <div className="search-container">
          <label>Search against README:</label>
          <input
            type="text"
            value={this.state.value}
            onChange={this.onChange} />
        </div>
        <div className="results-container">
          <Results results={results} />
        </div>
      </div>
```

```
    );
  }
  onChange(e) {
    const value = e.target.value;
    const index = this.state.index;
    const lines = this.state.lines;

    // Set captured value to input
    this.setState({
      value
    });

    // Search against lines and index if they exist
    if(lines && index) {
      this.setState({
        results: this.search(lines, index, value)
      });

      return;
    }

    // If the index doesn't exist, we need to set it up.
    // Unfortunately we cannot pass the path so we need to
    // hardcode it (Webpack uses static analysis).
    //
    // You could show loading indicator here as loading might
    // take a while depending on the size of the index.
    loadIndex().then(lunr => {
      // Search against the index now.
      this.setState({
        index: lunr.index,
        lines: lunr.lines,
        results: this.search(lunr.lines, lunr.index, value)
      });
    }).catch(err => {
      // Something unexpected happened (connection lost
      // for example).
      console.error(err);
    });
  }
  search(lines, index, query) {
    // Search against index and match README lines against the results.
```

```
    return index.search(query.trim()).map(match => lines[match.ref]);
  }
};

const Results = ({results}) => {
  if(results.length) {
    return (<ul>{
      results.map((result, i) => <li key={i}>{result}</li>)
    }</ul>);
  }

  return <span>No results</span>;
};

function loadIndex() {
  // Here's the magic. Set up `require.ensure` to tell Webpack
  // to split here and load our search index dynamically.
  //
  // The first parameter defines possible dependencies that
  // must be loaded first. Given there aren't any, we will
  // leave it as an empty array.
  return new Promise((resolve, reject) => {
    try {
      require.ensure([], require => {
        const lunr = require('lunr');
        const search = require('../search_index.json');

        resolve({
          index: lunr.Index.load(search.index),
          lines: search.lines
        });
      });
    }
    catch(err) {
      reject(err);
    }
  });
}
```

Even though there's a lot going on in this example, it's worth understanding. Webpack detect the `require.ensure` statically. It is able to generate a separate bundle based on this split point. Given it relies on static analysis, you cannot generalize `loadIndex` in this case and pass the search index path as a parameter.

The approach is useful with routers too. As the user enters some route, you can load the dependencies the resulting view needs. Alternatively you can start loading dependencies as the user scrolls a page and gets near parts with actual functionality. `require.ensure` provides a lot of power and allows you to keep your application lean.

`require.ensure` respects `output.publicPath` option.

Webpack 2 supports SystemJS[3] semantics. We can use `System.import('lunr').then(lunr => ...).catch(err => ...)` kind of declarations there.

There's a full example[4] showing how it all goes together with lunr, React, and Webpack.

20.3 Dynamic Loading with Webpack

Beyond `require.ensure`, there's another type of `require` that you should be aware of. It's require.context[5]. `require.context` is a type of `require` which contents aren't known compile-time.

Let's say you are writing a static site generator on top of Webpack. You could model your site contents within a directory structure. At the simplest level you could have just a `pages/` directory which would contain Markdown files.

Each of these files would have a YAML frontmatter for their metadata. The url of each page could be determined based on the filename. This is enough information to map the directory as a site. Code-wise we would end up with a statement like this somewhere:

[3] https://github.com/systemjs/systemjs
[4] https://github.com/survivejs/lunr-demo
[5] https://webpack.github.io/docs/context.html

```
// Process pages through `yaml-frontmatter-loader` and `json-loader`.
// The first one extracts the frontmatter and the body and the latter
// converts it into a JSON structure we can use later. Markdown hasn't
// been processed yet.
const req = require.context(
  'json!yaml-frontmatter!./pages',
  true, // Load files recursively. Pass false to skip recursion.
  /^\.\/.*\.md$/ // Match files ending with .md.
);
```

`require.context` returns us a function to `require` against. It also knows its module `id` and it provides a `keys()` method for figuring out the contents of the context. To give you a better example, consider the code below:

```
req.keys(); // ['./demo.md', './another-demo.md']

req.id; // 42

// {title: 'Demo', __content: '# Demo page\nDemo content\n\n'}
const demoPage = req('./demo.md');
```

This information is enough for generating an entire site. And this is exactly what I've done with Antwar[6]. You can find a more elaborate example in that static site generator.

The technique can be useful for other purposes, such as testing. When you need a dynamic require, `require.context` works.

Note that Webpack will also turn statements written in the form `require('./pages/' + pageName + '.md')` into the `require.context` format!

20.4 Conclusion

Understanding how Webpack's chunking works helps you to untap a lot of its power. Just applying `require.ensure` alone can be very effective. It opens a world of possibilities. `require.context` has more limited possibilities, but it's a powerful tool especially for tool developers.

[6] https://github.com/antwarjs/antwar

21. Linting in Webpack

Nothing is easier than making mistakes when coding in JavaScript. Linting is one of those techniques that can help you to make less mistakes. You can spot issues before they become actual problems.

Better yet, modern editors and IDEs offer strong support for popular tools. This means you can spot possible issues as you are developing. Despite this, it is a good idea to set them up with Webpack. That allows you to cancel a production build that might not be up to your standards for example.

21.1 Brief History of Linting in JavaScript

The linter that started it it all for JavaScript is Douglas Crockford's JSLint[1]. It is opinionated like the man himself. The next step in evolution was JSHint[2]. It took the opinionated edge out of JSLint and allowed for more customization. ESLint[3] is the newest tool in vogue.

ESLint goes to the next level as it allows you to implement custom rules, parsers, and reporters. ESLint works with Babel and JSX syntax making it ideal for React projects. The project rules have been documented well and you have full control over their severity. These features alone make it a powerful tool.

It is quite telling that a competing project, JSCS, decided to merge its efforts with ESLint[4]. JSCS reached end of life with its 3.0.0 release and the core team joined with ESLint.

Besides linting for issues, it can be useful to manage the code style on some level. Nothing is more annoying than having to work with source code that has mixed tabs and spaces. Stylistically consistent code reads better and is easier to work with. Linting tools allow you to do this.

21.2 Webpack and JSHint

Interestingly, no JSLint loader seems to exist for Webpack yet. Fortunately, there's one for JSHint. You could set it up on a legacy project easily. To get started, install jshint-loader[5] to your project first:

[1] http://www.jslint.com/
[2] http://jshint.com/
[3] http://eslint.org/
[4] http://eslint.org/blog/2016/04/welcoming-jscs-to-eslint
[5] https://www.npmjs.com/package/jshint-loader

```
npm i jshint jshint-loader --save-dev
```

In addition, you will need a little bit of configuration:

```
var common = {
  ...
  module: {
    preLoaders: [
      {
        test: /\.jsx?$/,
        loaders: ['jshint'],
        // define an include so we check just the files we need
        include: PATHS.app
      }
    ]
  },
};
```

`preLoaders` section of the configuration gets executed before `loaders`. If linting fails, you'll know about it first. There's a third section, `postLoaders`, that gets executed after `loaders`. You could include code coverage checking there during testing, for instance.

JSHint will look into specific rules to apply from `.jshintrc`. You can also define custom settings within a `jshint` object at your Webpack configuration. Exact configuration options have been covered at the JSHint documentation[6] in detail. `.jshintrc` could look like this:

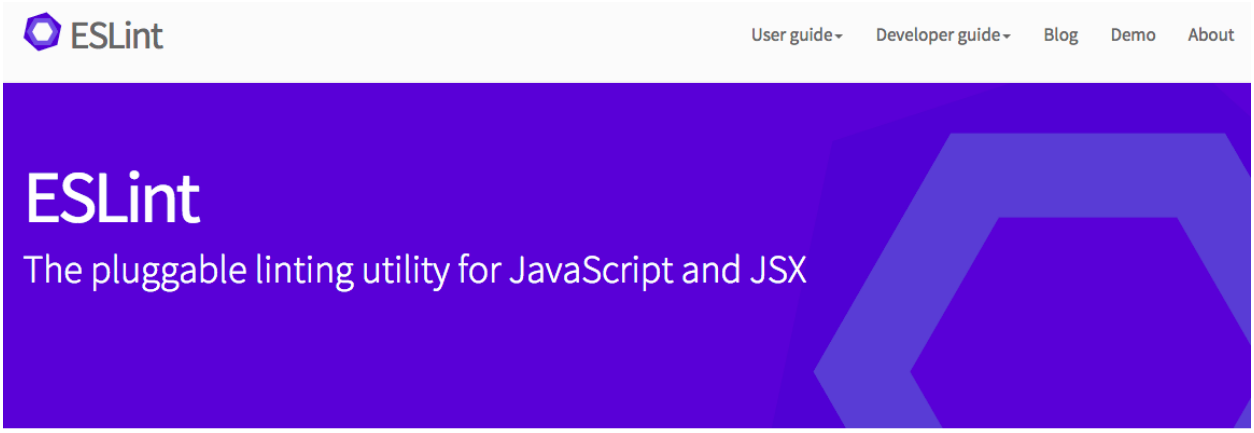
.jshintrc

```
{
  "browser": true,
  "camelcase": false,
  "esnext": true,
  "indent": 2,
  "latedef": false,
  "newcap": true,
  "quotmark": "double"
}
```

This tells JSHint we're operating within browser environment, don't care about linting for camelcase naming, want to use double quotes everywhere and so on.

[6] http://jshint.com/docs/

21.3 Setting Up ESLint



ESLint

ESLint[7] is a recent linting solution for JavaScript. It builds on top of ideas presented by JSLint and JSHint. More importantly it allows you to develop custom rules. As a result, a nice set of rules have been developed for React in the form of eslint-plugin-react[8].

Since *v1.4.0* ESLint supports a feature known as autofixing[9]. It allows you to perform certain rule fixes automatically. To activate it, pass the flag `--fix` to the tool. It is also possible to use this feature with Webpack, although you should be careful with it.

Connecting ESlint with *package.json*

To get started, install ESLint as a development dependency:

```
npm i eslint --save-dev
```

This will add ESLint and the loader we want to use as our project development dependencies. Next, we'll need to do some configuration so we can run ESLint easily through npm. I am using the `test` namespace to signify it's a testing related task. I am also enabling caching to improve performance on subsequent runs. Add the following:

package.json

[7] http://eslint.org/
[8] https://www.npmjs.com/package/eslint-plugin-react
[9] http://eslint.org/blog/2015/09/eslint-v1.4.0-released/

```
"scripts": {
  ...
  "test:lint": "eslint . --ext .js --ext .jsx --cache"
}
...
```

If you run `npm run test:lint` now, it will trigger ESLint against all JS and JSX files of our project. This configuration will likely lint a bit too much. Set up *.eslintignore* to the project root like this to skip *build/*:

.eslintignore

```
build/
```

Given most projects contain *.gitignore*, ESLint can be configured to use that instead of *.eslintignore* like this:

package.json

```
"scripts": {
  ...
  "test:lint": "eslint . --ext .js --ext .jsx --ignore-path .gitignore --cache"
}
...
```

If you need to ignore some specific directory in addition to the *.gitignore* definitions, you can pass `--ignore-pattern dist` kind of declaration to ESLint.

ESLint supports custom formatters through `--format` parameter. eslint-friendly-formatter[10] is an example of a formatter that provides terminal friendly output. This way you can jump conveniently straight to the warnings and errors from there.

You can get more performance out of ESLint by running it through a daemon, such as eslint_d[11]. Using it brings down the overhead and it can bring down linting times considerably.

[10] https://www.npmjs.com/package/eslint-friendly-formatter
[11] https://www.npmjs.com/package/eslint_d

Configuring ESLint

In order to truly benefit from ESLint, you'll need to configure it. There are a lot of rules included, you can load even more through plugins, and you can even write your own. See the official ESLint rules documentation[12] for more details on rules.

Consider the sample configuration below. It extends the recommended set of rules with some of our own:

.eslintrc

```
{
  // Extend existing configuration
  // from ESlint and eslint-plugin-react defaults.
  "extends": [
    "eslint:recommended", "plugin:react/recommended"
  ],
  // Enable ES6 support. If you want to use custom Babel
  // features, you will need to enable a custom parser
  // as described in a section below.
  "parserOptions": {
    "ecmaVersion": 6,
    "sourceType": "module"
  },
  "env": {
    "browser": true,
    "node": true
  },
  // Enable custom plugin known as eslint-plugin-react
  "plugins": [
    "react"
  ],
  "rules": {
    // Disable `no-console` rule
    "no-console": 0,
    // Give a warning if identifiers contain underscores
    "no-underscore-dangle": 1,
    // Default to single quotes and raise an error if something
    // else is used
    "quotes": [2, "single"]
  }
}
```

[12] http://eslint.org/docs/rules/

Beyond vanilla JSON, ESLint supports other formats, such as JavaScript or YAML. If you want to use a different format, name the file accordingly. I.e., *.eslintrc.js* would work for JavaScript. In that case you should export the configuration through `module.exports`. See the documentation[13] for further details.

It is possible to generate a custom *.eslintrc* by using `eslint --init`. It will ask you a series of questions and then write the file for you.

ESLint supports ES6 features through configuration. You will have to specify the features to use through the ecmaFeatures[14] property.

There are useful plugins, such as eslint-plugin-react[15] and eslint-plugin-import[16], that you might want to consider integrating to your project.

Connecting ESLint with Babel

In case you want to lint against custom language features that go beyond standard ES6, use babel-eslint[17]. Install the parser first:

```
npm i babel-eslint --save-dev
```

Change *.eslintrc* like this so that ESLint knows to use the custom parser over the default one:

```
{
  ...
  "parser": "babel-eslint",
  "parserOptions": {
    "ecmaVersion": 6,
    "sourceType": "module"
  },
  ...
}
```

[13] http://eslint.org/docs/user-guide/configuring#configuration-file-formats
[14] http://eslint.org/docs/user-guide/configuring.html#specifying-language-options
[15] https://www.npmjs.com/package/eslint-plugin-react
[16] https://www.npmjs.com/package/eslint-plugin-import
[17] https://www.npmjs.com/package/babel-eslint

Severity of ESLint Rules

The severity of an individual rule is defined by a number as follows:

- `0` - The rule has been disabled.
- `1` - The rule will emit a warning.
- `2` - The rule will emit an error.

Some rules, such as `quotes`, accept an array instead. This allows you to pass extra parameters to them. Refer to the rule's documentation for specifics.

Note that you can write ESLint configuration directly to *package.json*. Set up a `eslintConfig` field, and write your declarations below it.

It is possible to generate a sample `.eslintrc` using `eslint --init` (or `node_modules/.bin/eslint --init` for local install). This can be useful on new projects.

Dealing with `ELIFECYCLE` Error

In case the linting process fails, `npm` will give you a nasty looking `ELIFECYCLE` error. A good way to achieve a tidier output is to invoke `npm run lint --silent`. That will hide the `ELIFECYCLE` bit. You can define an alias for this purpose. In Unix you would do `alias run='npm run --silent'` and then `run <script>`.

Alternatively, you could pipe output to `true` like this:

package.json

```
"scripts": {
  ...
  "test:lint": "eslint . --ext .js --ext .jsx || true"
}
...
```

The problem with this approach is that if you invoke `test:lint` through some other command, it will pass even if there are failures. If you have another script that does something like `npm run test:lint && npm run build`, it will build regardless of the output of the first command!

Connecting ESLint with Webpack

We can make Webpack emit ESLint messages for us by using eslint-loader[18]. As the first step execute

[18] https://www.npmjs.com/package/eslint-loader

```
npm i eslint-loader --save-dev
```

Note that `eslint-loader` will use a globally installed version of ESLint unless you have one included with the project itself! Make sure you have ESLint as a development dependency to avoid strange behavior.

A good way to set it up is to go through `preLoaders` like this:

webpack.config.js

```
const common = {
  ...
  module: {
    preLoaders: [
      {
        test: /\.jsx?$/,
        loaders: ['eslint'],
        include: PATHS.app
      }
    ]
  },
  ...
};
```

By using a preloader definition, we make sure ESLint goes through our code before processing it any further. This way we'll know early of if our code fails to lint during development. It is useful to include linting to the production target of your project as well.

If you execute `npm start` now and break some linting rule while developing, you should see that in the terminal output. The same should happen when you build the project.

21.4 Customizing ESLint

Even though you can get very far with vanilla ESLint, there are several techniques you should be aware of. For instance, sometimes you might want to skip some particular rules per file. You might even want to implement rules of your own. We'll cover these cases briefly next.

Skipping ESLint Rules

Sometimes, you'll want to skip certain rules per file or per line. This can be useful when you happen to have some exceptional case in your code where some particular rule doesn't make sense. As usual, exception confirms the rule. Consider the following examples:

```
// everything
/* eslint-disable */
...
/* eslint-enable */


// specific rule
/* eslint-disable no-unused-vars */
...
/* eslint-enable no-unused-vars */


// tweaking a rule
/* eslint no-comma-dangle:1 */


// disable rule per line
alert('foo'); // eslint-disable-line no-alert
```

Note that the rule specific examples assume you have the rules in your configuration in the first place! You cannot specify new rules here. Instead, you can modify the behavior of existing rules.

Setting Environment

Sometimes, you may want to run ESLint in a specific environment, such as Node.js or Mocha. These environments have certain conventions of their own. For instance, Mocha relies on custom keywords (e.g., `describe`, `it`) and it's good if the linter doesn't choke on those.

ESLint provides two ways to deal with this: local and global. If you want to set it per file, you can use a declaration at the beginning of a file:

```
/*eslint-env node, mocha */
```

Global configuration is possible as well. In this case, you can use `env` key like this:

.eslintrc

```
{
  "env": {
    "browser": true,
    "node": true,
    "mocha": true
  },
  ...
}
```

Writing Your Own Rules

ESLint rules rely on Abstract Syntax Tree (AST) definition of JavaScript. It is a data structure that describes JavaScript code after it has been lexically analyzed. There are tools, such as recast[19], that allow you to perform transformations on JavaScript code by using AST transformations. The idea is that you match some structure, then transform it somehow and convert AST back to JavaScript.

To get a better idea of how AST works and what it looks like, you can check Esprima online JavaScript AST visualization[20] or AST Explorer by Felix Kling[21]. Alternatively you can install `recast` and examine the output it gives. That is the structure we'll be working with for ESLint rules.

Codemod[22] allows you to perform large scale changes to your codebase through AST based transformations.

In ESLint's case we just want to check the structure and report in case something is wrong. Getting a simple rule done is surprisingly simple:

1. Set up a new project named `eslint-plugin-custom`. You can replace `custom` with whatever you want. ESLint follows this naming convention.
2. Execute `npm init -y` to create a dummy *package.json*
3. Set up `index.js` in the project root with content like this:

eslint-plugin-custom/index.js

[19] https://github.com/benjamn/recast
[20] http://esprima.org/demo/parse.html
[21] http://astexplorer.net/
[22] https://github.com/facebook/codemod

```
module.exports = {
  rules: {
    demo: function(context) {
      return {
        Identifier: function(node) {
          context.report(node, 'This is unexpected!');
        }
      };
    }
  }
};
```

In this case, we just report for every identifier found. In practice, you'll likely want to do something more complex than this, but this is a good starting point.

Next, you need to execute `npm link` within `eslint-plugin-custom`. This will make your plugin visible within your system. `npm link` allows you to easily consume a development version of a library you are developing. To reverse the link you can execute `npm unlink` when you feel like it.

If you want to do something serious, you should point to your plugin through *package.json*.

We need to alter our project configuration to make it find the plugin and the rule within.

.eslintrc

```
{
  ...
  "plugins": [
~~    "react",~~
    "react",
    "custom"
  ],
  "rules": {
    "custom/demo": 1,
    ...
  }
}
```

If you invoke ESLint now, you should see a bunch of warnings. Mission accomplished!

Of course the rule doesn't do anything useful yet. To move forward, I recommend checking out the official documentation about plugins[23] and rules[24].

You can also check out some of the existing rules and plugins for inspiration to see how they achieve certain things. ESLint allows you to extend these rulesets[25] through `extends` property. It accepts either a path to it (`"extends": "./node_modules/coding-standard/.eslintrc"`) or an array of paths. The entries are applied in the given order and later ones override the former.

ESLint Resources

Besides the official documentation available at eslint.org[26], you should check out the following blog posts:

- Lint Like It's 2015[27] - This post by Dan Abramov shows how to get ESLint to work well with Sublime Text.
- Detect Problems in JavaScript Automatically with ESLint[28] - A good tutorial on the topic.
- Understanding the Real Advantages of Using ESLint[29] - Evan Schultz's post digs into details.
- eslint-plugin-smells[30] - This plugin by Elijah Manor allows you to lint against various JavaScript smells. Recommended.

If you just want some starting point, you can pick one of eslint-config- packages[31] or go with the standard[32] style. By the looks of it, `standard` has some issues with JSX[33] so be careful with that.

21.5 Linting CSS

stylelint[34] allows us to lint CSS. It can be used with Webpack through postcss-loader[35].

```
npm i stylelint postcss-loader --save-dev
```

Next, we'll need to integrate it with our configuration:

webpack.config.js

[23] http://eslint.org/docs/developer-guide/working-with-plugins.html
[24] http://eslint.org/docs/developer-guide/working-with-rules.html
[25] http://eslint.org/docs/user-guide/configuring.html#extending-configuration-files
[26] http://eslint.org/
[27] https://medium.com/@dan_abramov/lint-like-it-s-2015-6987d44c5b48
[28] http://davidwalsh.name/eslint
[29] http://rangle.io/blog/understanding-the-real-advantages-of-using-eslint/
[30] https://github.com/elijahmanor/eslint-plugin-smells
[31] https://www.npmjs.com/search?q=eslint-config
[32] https://www.npmjs.com/package/standard
[33] https://github.com/feross/standard/issues/138
[34] http://stylelint.io/
[35] https://www.npmjs.com/package/postcss-loader

```
...
const stylelint = require('stylelint');

...

const common = {
  ...
  module: {
    preLoaders: [
      {
        test: /\.css$/,
        loaders: ['postcss'],
        include: PATHS.app
      },
      ...
    ],
    ...
  },
  postcss: function () {
    return [
      stylelint({
        rules: {
          'color-hex-case': 'lower'
        }
      })
    ];
  },
  ...
}
```

If you define a CSS rule, such as `background-color: #EFEFEF;`, you should see a warning at your terminal. See stylelint documentation for a full list of rules. npm lists possible stylelint rulesets[36]. You consume them as your project dependency like this:

```
const configSuitcss = require('stylelint-config-suitcss');

...

stylelint(configSuitcss)
```

It is possible to define configuration through a *.stylelintrc* file. The idea is similar as for other linting tools. There's also a CLI available.

[36] https://www.npmjs.com/search?q=stylelint-config

If you want to try out an alternative way to set up stylelint, consider using the stylelint-webpack-plugin[37] instead.

21.6 EditorConfig

EditorConfig[38] allows you to maintain a consistent coding style across different IDEs and editors. Some even come with built-in support. For others, you have to install a separate plugin. In addition to this you'll need to set up a `.editorconfig` file like this:

.editorconfig

```
root = true

# General settings for whole project
[*]
indent_style = space
indent_size = 4

end_of_line = lf
charset = utf-8
trim_trailing_whitespace = true
insert_final_newline = true

# Format specific overrides
[*.md]
trim_trailing_whitespace = false

[app/**.js]
indent_style = space
indent_size = 2
```

21.7 Conclusion

In this chapter, you learned how to lint your code using Webpack in various ways. It is one of those techniques that yields benefits over the long term. You can fix possible problems before they become actual issues.

[37] https://www.npmjs.com/package/stylelint-webpack-plugin
[38] http://editorconfig.org/

22. Authoring Packages

Even though Webpack is useful for bundling applications, it has its uses for package authors as well. You can use it to output your bundle in the UMD format[1]. It is a format that's compatible with various environments (CommonJS, AMD, globals).

As Webpack alone isn't enough, I'll provide a short overview of the npm side of things before discussing specific techniques.

22.1 Anatomy of a npm Package

Most of the available npm packages are small and include just a select few files:

- *index.js* - On small projects it's enough to have the code at the root. On larger ones you may want to start splitting it up further.
- *package.json* - npm metadata in JSON format
- *README.md* - README is the most important document of your project. It is written in Markdown format and provides an overview. For simple projects the whole documentation can fit there. It will be shown at the package page at *npmjs.com*.
- *LICENSE* - You should include licensing information within your project. You can refer to it from *package.json*.

In larger projects, you may find the following:

- *CONTRIBUTING.md* - A guide for potential contributors. How should the code be developed and so on.
- *CHANGELOG.md* - This document describes major changes per version. If you do major API changes, it can be a good idea to cover them here. It is possible to generate the file based on Git commit history, provided you write nice enough commits.
- *.travis.yml* - Travis CI[2] is a popular continuous integration platform that is free for open source projects. You can run the tests of your package over multiple systems using it. There are other alternatives of course, but Travis is very popular.
- *.gitignore* - Ignore patterns for Git, i.e., which files shouldn't go under version control. It can be useful to ignore npm distribution files here so they don't clutter your repository.

[1] https://github.com/umdjs/umd
[2] https://travis-ci.org/

- *.npmignore* - Ignore patterns for npm. This describes which files shouldn't go to your distribution version. A good alternative is to use the files[3] field at *package.json*. It allows you to maintain a whitelist of files to include into your distribution version.
- *.eslintignore* - Ignore patterns for ESLint. Again, tool specific.
- *.eslintrc* - Linting rules. You can use *.jshintrc* and such based on your preferences.
- *webpack.config.js* - If you are using a simple setup, you might as well have the configuration at project root.

In addition, you'll likely have various directories for source, tests, demos, documentation, and so on.

If you want to decrease the size of your dependencies, consider using a tool like package-config-checker[4]. It can pinpoint packages not using the `files` field correctly. Once you know which ones haven't set it, you can consider making Pull Requests (PRs) to those projects.

22.2 Understanding *package.json*

All packages come with a *package.json* that describes metadata related to them. This includes information about the author, various links, dependencies, and so on. The official documentation[5] covers them in detail.

I've annotated a part of *package.json* of my React component boilerplate[6] below:

```
{
  /* Name of the project */
  "name": "react-component-boilerplate",
  /* Brief description */
  "description": "Boilerplate for React.js components",
  /* Who is the author + optional email + optional site */
  "author": "Juho Veps□l□inen <email goes here> (site goes here)",
  /* Version of the package */
  "version": "0.0.0",
  /* `npm run <name>` */
  "scripts": {
    "start": "webpack-dev-server",
```

[3] https://docs.npmjs.com/files/package.json#files
[4] https://www.npmjs.com/package/package-config-checker
[5] https://docs.npmjs.com/files/package.json
[6] https://github.com/survivejs/react-component-boilerplate

```
    "test": "karma start",
    "test:tdd": "npm run test -- --auto-watch --no-single-run",
    "test:lint": "eslint . --ext .js --ext .jsx --cache",

    "gh-pages": "webpack",
    "gh-pages:deploy": "gh-pages -d gh-pages",
    "gh-pages:stats": "webpack --profile --json > stats.json",

    "dist": "webpack",
    "dist:min": "webpack",
    "dist:modules": "rm -rf ./dist-modules && babel ./src --out-dir ./dist-modul\
es",

    "pretest": "npm run test:lint",
    "preversion": "npm run test && npm run dist && npm run dist:min && git commi\
t --allow-empty -am \"Update dist\"",
    "prepublish": "npm run dist:modules",
    "postpublish": "npm run gh-pages && npm run gh-pages:deploy",
    /* If your library is installed through Git, you may want to transpile it */
    "postinstall": "node lib/post_install.js"
  },
  /* Entry point for terminal (i.e., <package name>) */
  /* Don't set this unless you intend to allow CLI usage */
  "bin": "./index.js",
  /* Entry point (defaults to index.js) */
  "main": "dist-modules",
  /* Package dependencies */
  "dependencies": {},
  /* Package development dependencies */
  "devDependencies": {
    "babel": "^6.3.17",
    ...
    "webpack": "^1.12.2",
    "webpack-dev-server": "^1.12.0",
    "webpack-merge": "^0.7.0"
  },
  /* Package peer dependencies. The consumer will fix exact versions. */
  /* In npm3 these won't get installed automatically and it's up to the */
  /* user to define which versions to use. */
  /* If you want to include RC versions to the range, consider using */
  /* a pattern such as ^4.0.0-0 */
  "peerDependencies": {
```

```
    "lodash": ">= 3.5.0 < 4.0.0",
    "react": ">= 0.11.2 < 16.0.0"
  }
  /* Links to repository, homepage, and issue tracker */
  "repository": {
    "type": "git",
    "url": "https://github.com/bebraw/react-component-boilerplate.git"
  },
  "homepage": "https://bebraw.github.io/react-component-boilerplate/",
  "bugs": {
    "url": "https://github.com/bebraw/react-component-boilerplate/issues"
  },
  /* Keywords related to package. */
  /* Fill this well to make the package findable. */
  "keywords": [
    "react",
    "reactjs",
    "boilerplate"
  ],
  /* Which license to use */
  "license": "MIT"
}
```

As you can see, *package.json* can contain a lot of information. You can attach non-npm specific metadata there that can be used by tooling. Given this can bloat *package.json*, it may be preferable to keep metadata at files of their own.

JSON doesn't support comments even though I'm using them above. There are extended notations, such as Hjson[7], that do.

22.3 npm Workflow

Working with npm is surprisingly simple. To get started, you will need to use npm adduser[8] (aliased to `npm login`). It allows you to set up an account. After this process has completed, it will create *~/.npmrc* and use that data for authentication. There's also npm logout[9] that will clear the credentials.

[7] http://hjson.org/

[8] https://docs.npmjs.com/cli/adduser

[9] https://docs.npmjs.com/cli/logout

When creating a project, `npm init` respects the values set at *~/.npmrc*. Hence it may be worth your while to set reasonable defaults there to save some time.

Publishing a Package

Provided you have logged in, creating new packages is just a matter of executing `npm publish`. Given that the package name is still available and everything goes fine, you should have something out there! After this, you can install your package through `npm install` or `npm i`.

An alternative way to consume a library is to point at it directly in *package.json*. In that case, you can do `"depName": "<github user>/<project>#<reference>"` where `<reference>` can be either commit hash, tag, or branch. This can be useful, especially if you need to hack around something and cannot wait for a fix.

If you want to see what files will be published to npm, consider using a tool known as irish-pub[10]. It will give you a listing to review.

Bumping a Version

In order to bump your package version, you'll just need to invoke one of these commands:

- `npm version <x.y.z>` - Define version yourself.
- `npm version <major|minor|patch>` - Let npm bump the version for you based on SemVer.
- `npm version <premajor|preminor|prepatch|prerelease>` - Same as previous expect this time it will generate `-<prerelease number>` suffix. Example: `v2.1.2-2`.

Invoking any of these will update *package.json* and create a version commit to git automatically. If you execute `npm publish` after doing this, you should have something new out there.

Note that in the example above I've set up `version` related hooks to make sure a version will contain a fresh version of a distribution build. I also run tests just in case. It's better to catch potential issues early on after all.

Consider using semantic-release[11] if you prefer more structured approach. It can take some pain out of the release process while automating a part of it. For instance, it is able to detect possible breaking changes and generate changelogs.

[10] https://www.npmjs.com/package/irish-pub
[11] https://www.npmjs.com/package/semantic-release

Respect the SemVer

Even though it is simple to publish new versions out there, it is important to respect the SemVer. Roughly, it states that you should not break backwards compatibility, given certain rules are met. For example, if your current version is `0.1.4` and you do a breaking change, you should bump to `0.2.0` and document the changes. You can understand SemVer much better by studying the online tool[12] and how it behaves.

Publishing a Prerelease Version

Sometimes, you might want to publish something preliminary for other people to test. There are certain conventions for this. You rarely see *alpha* releases at npm. *beta* and *rc (release candidate) are common, though. For example, a package might have versions like this:

- v0.5.0-alpha1
- v0.5.0-beta1
- v0.5.0-beta2
- v0.5.0-rc1
- v0.5.0-rc2
- v0.5.0

The initial alpha release will allow the users to try out the upcoming functionality and provide feedback. The beta releases can be considered more stable. The release candidates (rc) are close to an actual release and won't introduce any new functionality. They are all about refining the release till it's suitable for general consumption.

The workflow in this case is straight-forward:

1. `npm version 0.5.0-alpha1` - Update *package.json* as discussed earlier.
2. `npm publish --tag alpha1` - Publish the package under *alpha1* tag.

In order to consume the test version, your users will have to use `npm i <your package name>@alpha1`.

It can be useful to utilize `npm link` during development. That will allow you to use a development version of your library from some other context. Node.js will resolve to the linked version unless local `node_modules` happens to contain a version. If you want to remove the link, use `npm unlink`.

[12] http://semver.npmjs.com/

On Naming Packages

Before starting to develop, it can be a good idea to spend a little bit of time on figuring out a good name for your package. It's not very fun to write a great package just to notice the name has been taken. A good name is easy to find through a search engine, and most importantly, is available at npm.

As of npm 2.7.0 it is possible to create scoped packages[13]. They follow format `@username/project-name`. Simply follow that when naming your project.

Version Ranges

npm supports multiple version ranges. I've listed the common ones below:

- ~ - Tilde matches only patch versions. For example, `~1.2` would be equal to `1.2.x`.
- ^ - Caret is the default you get using `--save` or `--save-dev`. It matches to It matches minor versions. This means `^0.2.0` would be equal to `0.2.x`.
- * - Asterisk matches major releases. This is the most dangerous of the ranges. Using this recklessly can easily break your project in the future and I would advise against using it.
- `>= 1.3.0 < 2.0.0` - Range between versions. This can be particularly useful if you are using `peerDependencies`.

You can set the default range using `npm config set save-prefix='^'` in case you prefer something else than caret. Alternatively you can modify *~/.npmrc* directly. Especially defaulting to tilde can be a good idea that can help you to avoid some trouble with dependencies.

Sometimes, using version ranges can feel a little dangerous. What if some future version is broken? npm shrinkwrap[14] allows you to fix your dependency versions and have stricter control over the versions you are using in a production environment. lockdown[15] goes further and gives guarantees about dependency content, not just version.

If you want to bundle some dependencies with your distribution version, consider using the bundledDependencies[16] (or just `bundleDependencies`) field. This can be useful if you want to share third party files not available through npm. There's a great Stack Overflow answer[17] discussing the topic further.

[13] https://docs.npmjs.com/getting-started/scoped-packages
[14] https://docs.npmjs.com/cli/shrinkwrap
[15] https://www.npmjs.com/package/lockdown
[16] https://docs.npmjs.com/files/package.json#bundleddependencies
[17] http://stackoverflow.com/a/25044361/228885

22.4 npm Lifecycle Hooks

npm provides various lifecycle hooks that can be useful. Suppose you are authoring a React component using Babel and some of its goodies. You could let the *package.json* `main` field point at the UMD version as generated above. This won't be ideal for those consuming the library through npm, though.

It is better to generate a ES5 compatible version of the package for npm consumers. This can be achieved using **babel** CLI tool:

```
babel ./lib --out-dir ./dist-modules
```

This will walk through the `./lib` directory and output a processed file for each library it encounters to `./dist-modules`.

Since we want to avoid having to run the command directly whenever we publish a new version, we can connect it to `prepublish` hook like this:

```
"scripts": {
  ...
  "prepublish": "babel ./lib --out-dir ./dist-modules"
}
```

Make sure you execute `npm i babel --save-dev` to include the tool into your project.

You probably don't want the directory content to end up in your Git repository. In order to avoid this and to keep your `git status` clean, consider this sort of `.gitignore`:

```
dist-modules/
...
```

Besides `prepublish`, npm provides a set of other hooks. The naming is always the same and follows the pattern `pre<hook>`, `<hook>`, `post<hook>` where `<hook>` can be `publish`, `install`, `test`, `stop`, `start`, `restart`, or `version`. Even though npm will trigger scripts bound to these automatically, you can trigger them explicitly through `npm run` for testing (i.e., `npm run prepublish`).

There are plenty of smaller tricks to learn for advanced usage. Those are better covered by the official documentation[18]. Often all you need is just a `prepublish` script for build automation.

[18] https://docs.npmjs.com/misc/scripts

22.5 Keeping Dependencies Up to Date

An important part of maintaining npm packages is keeping their dependencies up to date. How to do this depends a lot on the maturity of your package. Ideally, you have a nice set of tests covering the functionality. If not, things can get a little hairier. There are a few ways to approach dependency updates:

- You can update all dependencies at once and hope for the best. Tools, such as npm-check-updates[19] or npm-check[20], can do this for you.
- Install the newest version of some specific dependency, e.g., `npm i lodash@* --save`. This is a more controlled way to approach the problem.
- Patch version information by hand by modifying *package.json* directly.

It is important to remember that your dependencies may introduce backwards incompatible changes. It can be useful to remember how SemVer works and study dependency release notes. They might not always exist, so you may have to go through the project commit history. There are a few services that can help you to keep track of your project dependencies:

- David[21]
- versioneye[22]
- Gemnasium[23]

These services provide badges you can integrate into your project *README.md* and they may email you about important changes. They can also point out possible security issues that have been fixed.

For testing your projects you can consider solutions, such as Travis CI[24] or SauceLabs[25]. Coveralls[26] gives you code coverage information and a badge to include in your README.

The services are valuable as they allow you to test your updates against a variety of platforms quickly. Something that might work on your system might not work in some specific configuration. You'll want to know about that as fast as possible to avoid introducing problems.

shields.io[27] lists a large amount of available badges.

[19] https://www.npmjs.com/package/npm-check-updates
[20] https://www.npmjs.com/package/npm-check
[21] https://david-dm.org/
[22] https://www.versioneye.com/
[23] https://gemnasium.com
[24] https://travis-ci.org/
[25] https://saucelabs.com/
[26] https://coveralls.io/
[27] http://shields.io/

22.6 Sharing Authorship

As packages evolve, you may want to start developing with others. You could become the new maintainer of some project, or pass the torch to someone else. These things happen as packages evolve.

npm provides a few commands for these purposes. It's all behind `npm owner` namespace. More specifically, you'll find `ls <package name>`, `add <user> <package name>` and `rm <user> <package name>` there (i.e., `npm owner ls`). That's about it.

See npm documentation[28] for the most up to date information about the topic.

22.7 Package Authoring Techniques

There are a couple of package authoring related techniques that are good to know. You can set up Webpack to generate a UMD build. You can also exclude certain dependencies out of your bundle. To make it easier to consume your packages, you can also generate a Node.js friendly versions. This technique can be improved further by setting up a script to generate a Node.js friendly version.

Setting Up UMD

Allowing Webpack to output your bundle in the UMD format is simple. Webpack allows you to control the output format using output.libraryTarget[29] field. It defaults to `var`. This means it will set your bundle to a variable defined using the `output.library` field.

There are other options too, but the one we are interested in is `output.libraryTarget: 'umd'`. Consider the example below:

webpack.config.js

```
output: {
  path: PATHS.dist,
  libraryTarget: 'umd', // !!
  // Name of the generated global.
  library: 'MyLibrary',
  // Optional name for the generated AMD module.
  umdNamedDefine: 'my_library'
}
```

[28] https://docs.npmjs.com/cli/owner

[29] https://webpack.github.io/docs/configuration.html#output-librarytarget

Avoiding Bundling Dependencies

Given it's not a good idea to bundle your package dependencies, such as React, within the distribution bundle itself, you should let the user inject them. You can configure external dependencies using the `externals` configuration. You can control it like this:

webpack.config.js

```
externals: {
  // Adapt `import merge from 'lodash/merge';` to different environments.
  'lodash/merge': {
    commonjs: 'lodash/merge',
    commonjs2: 'lodash/merge',
    // Look up lodash.merge below ['lodash', 'merge'] for AMD.
    amd:  ['lodash', 'merge'],
    // Look up lodash.merge through `_.merge` in global environment.
    root: ['_', 'merge']
  },
  // Adapt React to different environments.
  'react': {
    commonjs: 'react',
    commonjs2: 'react',
    amd: 'React',
    root: 'React'
  }
},
```

These two fields help you a lot as a package author but there's more to it.

If you want to include all modules in *node_modules* by default, it could be interesting to use webpack-node-externals[30] instead. Then you would end up with `externals: [nodeExternals()]` kind of declaration. If you don't need to adapt to different environments, this could be a neat way to go.

Processing Node.js Version through Babel

If you are processing your code through Babel, I suggest you process the Node.js version of the package directly through Babel and skip Webpack. The advantage of doing this is that it gives you separate modules that are easier to consume one by one if needed. This avoids having to go through a heavy bundle. In this case you'll likely want a setup like this:

package.json

[30] https://www.npmjs.com/package/webpack-node-externals

```
{
  ..
  /* `npm run <name>` */
  "scripts": {
    ...
    "dist": "webpack",
    "dist:min": "webpack",

    /* Process source through Babel! */
    "dist:modules": "babel ./src --out-dir ./dist-modules",

    ...

    "preversion": "npm run test && npm run dist && npm run dist:min && git commi\
t --allow-empty -am \"Update dist\"",
    "prepublish": "npm run dist:modules",
    ...
  },
  /* Point to the Node.js specific version */
  "main": "dist-modules",
  ...
}
```

There is one problem, though. What if someone points to a development version of your package directly through GitHub? It simply won't work as the `dist-modules` directory will be missing. This can be fixed using a hook that will generate the needed source.

Generating a Distribution for Development Usage

To solve the development distribution problem, we need to hook up a custom script the right way. First, we need to connect the hook with a custom script:

package.json

```
{
  ...
  "scripts": {
    ...
    /* Point to the script that generates the missing source. */
    "postinstall": "node lib/post_install.js"
  },
  ...
}
```

Secondly we'll need the script itself:

lib/post_install.js

```
#!/usr/bin/env node
// adapted based on rackt/history (MIT)
const spawn = require('child_process').spawn;
const stat = require('fs').stat;

stat('dist-modules', function(error, stat) {
  if (error || !stat.isDirectory()) {
    spawn(
      'npm',
      [
        'i',
        'babel-cli',
        'babel-preset-es2015',
        'babel-preset-react'
      ],
      {
        stdio: [0, 1, 2]
      }
    ).on('close', function(exitCode) {
      spawn('npm', ['run', 'dist-modules'], { stdio: [0, 1, 2] });
    });
  }
});
```

The script may need tweaking to fit your purposes. But it's enough to give you a rough idea. If the `dist_modules` directory is missing, we'll generate it here. That's it.

Relying on `postinstall` scripts can be potentially dangerous[31]. Security minded developers may want to use `npm install --ignore-scripts`. You can set that default through `npm config set ignore-scripts true` if you want. Being a little cautious might not hurt.

22.8 Conclusion

You should now have a basic idea of how to author npm packages. Webpack can help you a lot here. Just picking up `output.libraryTarget` and `externals` help you a lot. These options are useful beyond package authoring. Particularly `externals` comes in handy when you want to exclude certain dependencies outside of your bundles and load them using some other way.

[31] http://blog.npmjs.org/post/141702881055/package-install-scripts-vulnerability

23. Writing Loaders

As we've seen so far, loaders are one of the building blocks of Webpack. If you want to load an asset, you'll most likely need to set up a matching loader definition. Even though there are a lot of available loaders[1], it is possible you are missing one fitting your purposes.

The official documentation[2] covers the loader API fairly well. To give you a concrete example, I'm going to discuss a subset of a loader I have developed. highlight-loader[3] accepts HTML and then applies highlight.js[4] on it. Even though the transformation itself is quite simple, the loader implementation isn't trivial.

23.1 Setting Up a Loader Project

I follow the following layout in my loader project:

```
.
├── LICENSE
├── README.md
├── demo
│   ├── demo_index.js
│   ├── input.md
│   ├── output
│   └── run_demo.js
├── index.js
├── node_modules
├── package.json
└── test.js
```

This is a fairly standard way to write a small Node.js package. `index.js` contains the loader source, `test.js` contains my tests, `demo` directory contains something that can be run against Webpack. I actually started by developing the demo first and added tests later on.

Writing tests first can be a good idea, though, as it gives you a specification which you can use to validate your implementation. It takes some advanced knowledge to pull this off, though. I'll give you a basic testing setup next and then discuss my loader implementation.

[1] https://webpack.github.io/docs/list-of-loaders.html
[2] https://webpack.github.io/docs/loaders.html
[3] https://github.com/bebraw/highlight-loader
[4] https://highlightjs.org/

23.2 Writing Tests for a Loader

I settled with Mocha[5] and Node.js assert[6] for this project. Mocha is nice as it provides just enough structure for writing your tests. There's also support for `--watch`. When you run Mocha in the watch mode, it will run the tests as your code evolves. This can be a very effective way to develop code.

Test Setup

The test setup itself is minimal. Here's the relevant *package.json* portion:

package.json

```
...
"scripts": {
  "test": "mocha ./test",
  "test:watch": "mocha ./test --watch"
},
...
```

To run tests, you can simply invoke `npm test`. To run the test setup in the watch mode, you can use `npm run test:watch`.

Test Structure

Given this is so small project, I ended up writing all my tests into a single file. The following excerpt should give you a better idea of what they look like. There are a couple of Webpack loader specific tweaks in place to make it easier to test them:

test.js

```
const assert = require('assert');
const assign = require('object-assign');
const loader = require('./');

// Mock loader context (`this`) so that we have an environment
// that's close enough to Webpack in order to avoid crashes
// during testing. Alternatively we could code defensively
// and protect against the missing data.
const webpackContext = {
  cacheable: noop,
```

[5] https://mochajs.org/

[6] https://nodejs.org/api/assert.html

```
  exec: noop
};
// Bind the context. After this we can run the loader in our
// tests.
const highlight = loader.bind(webpackContext);

// Here's the actual test suite split into smaller units.
describe('highlight-loader', function () {
  it('should highlight code', function () {
    const code = '<code>run &lt;script&gt;</code>';
    const given = highlight(code);
    const expected = '<code>run &lt;script&gt;</code>';

    assert.equal(given, expected);
  });

  ...

  it('should support raw output with lang', function () {
    const code = 'a = 4';
    // Pass custom query to the loader. In order to do this
    // we need to tweak the context (`this`).
    const given = loader.call(assign({}, webpackContext, {
      query: '?raw=true&lang=python'
    }), code);
    const expected = 'module.exports = ' +
      '"a = <span class=\\"hljs-number\\">4</span>"';

    assert.equal(given, expected);
  });

  ...
});

function noop() {}
```

Even though I'm not a great fan of mocking, it works well enough for a case like this. The biggest fear is that Webpack API changes at some point. This would mean my test code would break and I would have to rewrite a large part of it.

It could be interesting to run the tests through Webpack itself to avoid mocking. In this approach you wouldn't have to worry about the test facing parts so much and it would be more about capturing output for the given input. The problem is that this would add a significant overhead to the tests

and bring problems of its own as you would have to figure out more effective ways to execute them.

23.3 Implementing a Loader

The loader implementation itself isn't entirely trivial due to the amount of functionality within it. I use cheerio[7] to apply *highlight.js* on the code portions of the passed HTML. Cheerio provides an API resembling jQuery making it ideal for small tasks, such as this.

To keep this discussion simple, I'll give you a subset of the implementation next to show you the key parts:

```
'use strict';
...
var hl = require('highlight.js');
var loaderUtils = require('loader-utils');
var highlightAuto = hl.highlightAuto;
var highlight = hl.highlight;

module.exports = function(input) {
  // Failsafe against `undefined` as highlight.js
  // seems to fail with that.
  input = input || '';

  // Mark the loader as cacheable (same result for same input).
  this.cacheable();

  // Parse custom query parameters.
  var query = loaderUtils.parseQuery(this.query);

  // Check against a custom parameter and apply custom logic
  // related to it. In this case we execute against the parameter
  // itself and tweak `input` based on the result.
  if(query.exec) {
    // `this.resource` refers to the resource we are trying to load
    // while including the query parameter.
    input = this.exec(input, this.resource);
  }

  ...

  // Cheerio logic goes here.
```

[7] https://www.npmjs.org/package/cheerio

```
  ...

  // Return a result after the transformation has been done.
  return $.html();
};

...
```

This is an example of a synchronous loader. Sometimes you might want to perform asynchronous operations instead. That's when you could do something like this in your loader code:

```
//
var callback = this.async();

if(!callback) {
  // Synchronous fallback.
  return syncOp();
}

// Perform the asynchronous operation.
asyncOp(callback);
```

23.4 Conclusion

Writing loaders is fun in sense that they just describe transformations from a format to another. Often you can figure out how to achieve something specific by either studying the API documentation or existing loaders.

I recommend writing at least basic tests and a small demo to document your assumptions. Loader development fits this type of thinking very well.

24. Configuring React

Facebook's React[1] is a popular alternative for developing web applications. Even if you don't use it, it can be valuable to understand how to configure it. Most React setups rely on a transpiler known as Babel[2].

Babel is useful beyond React development and worth understanding as it allows you to use future JavaScript features now without having to worry too much about browser support. Due to technical constraints it doesn't support all features within the specification, but still it can be a good tool to have in your arsenal.

24.1 Setting Up Babel with React

Most of the React code out there relies on a format known as JSX[3]. It is a superset of JavaScript that allows you to mix XMLish syntax with JavaScript.

A lot of people find this convenient as they get something that resembles what they know already while they can use the power of JavaScript. This is in contrast to template DSLs that implement the same logic using custom constructs.

Some React developers prefer to attach type annotations to their code using a language extension known as Flow[4]. The technology fits React well, but it's not restricted to it. TypeScript[5] is another viable alternative. Both work with JSX.

Babel allows us to use JSX with React easily. In addition, we can enable language features we want either using plugins or presets that encapsulate collections of plugins. For instance you can find all ES6 features within a preset. The same goes for React related functionality. We'll be relying on these within our setup.

It is a good practice to name React components containing JSX using the `.jsx` suffix. In addition to communicating this fact, your editor can apply syntax highlighting automatically as you open the files.

[1] https://facebook.github.io/react/
[2] https://babeljs.io/
[3] https://facebook.github.io/jsx/
[4] http://flowtype.org/
[5] http://www.typescriptlang.org/

Installing *babel-loader*

The first step towards configuring Babel to work with Webpack is to set up babel-loader[6]. It will take our futuristic code and turn it into a format normal browsers can understand. Install *babel-loader with:

```
npm i babel-loader babel-core --save-dev
```

babel-core contains the core logic of Babel so we need to install that as well.

Connecting *babel-loader* with Webpack

Now that we have the loader installed, we can connect it with Webpack configuration. In addition to a loader definition, we can perform an additional tweak to make imports without an extension possible. Leaving the extension visible is a valid alternative.

Webpack provides a field known as resolve.extensions[7] that can be used for this purpose. If you want to allow imports like `import Button from './Button';`, set it up as follows:

webpack.config.js

```
...
const common = {
  ...
  // Important! Do not remove ''. If you do, imports without
  // an extension won't work anymore!
  resolve: {
    extensions: ['', '.js', '.jsx']
  }
}
...
```

The loader configuration is straight-forward as well. We can use a RegExp to match both `.js` and `.jsx` files. It's up to your tastes to figure out a neat pattern. I prefer to use `\.jsx?$` myself. This just makes `x` optional.

Alternatively, you could spell out the options using a matcher such as `\.(js|jsx)$`. Latter format can be particularly useful if you have to match against multiple different formats.

babel-loader comes with a set of options. In this case I'm going to enable `cacheDirectory` to improve its performance during development. Simply passing it as a flag helps. You can also pass a specific directory to it as a parameter. I.e., `babel?cacheDirectory=<path>`. Here's the full loader configuration:

webpack.config.js

[6] https://www.npmjs.com/package/babel-loader

[7] https://webpack.github.io/docs/configuration.html#resolve-extensions

```
...
module: {
  loaders: [
    {
      test: /\.jsx?$/,
      // Enable caching for improved performance during development
      // It uses default OS directory by default. If you need
      // something more custom, pass a path to it.
      // I.e., babel?cacheDirectory=<path>
      loaders: ['babel?cacheDirectory'],
      // Parse only app files! Without this it will go through
      // the entire project. In addition to being slow,
      // that will most likely result in an error.
      include: PATHS.app
    },
    ...
  }
}
...
```

Even though we have Babel installed and set up, we are still missing one bit - Babel configuration. It would be possible to pass it through the loader definition. Instead, I prefer to handle it using a dotfile known as *.babelrc*.

The benefit of doing this is that it allows us to process our Webpack configuration itself using the same language rules. The rules will also be shared with Babel running outside of Webpack. This can be useful for package authors.

If you want to process your Webpack configuration through Babel, name your Webpack configuration as *webpack.config.babel.js*. Webpack will notice you want to use Babel and execute your configuration through it.

Setting Up *.babelrc*

A minimal Babel and React setup needs just two Babel presets. Install them:

```
npm i babel-preset-es2015 babel-preset-react --save-dev
```

Instead of typing it all out, we could use brace expansion. Example: `npm i babel-preset-{es2015,react} -D`. `-D` equals `--save-dev` as you might remember. Note that this doesn't work on Windows CMD.

To make Babel aware of them, we need to write a *.babelrc*:

.babelrc

```
{
  "presets": [
    "es2015",
    "react"
  ]
}
```

Babel should pick up the presets now and you should be able to develop both ES6 and React code using Webpack now.

Sometimes you might want to use experimental features. Although you can find a lot of them within so called stage presets, I recommend enabling them one by one and even organizing them to a preset of their own unless you are working on a throwaway project. If you expect your project to live a long time, it's better to document the features you are using well.

There are other possible .babelrc options[8] beyond the ones covered here.

24.2 Setting Up Hot Loading

One of the features that sets React and Webpack apart is a feature known as hot loading. This is something that sits on top of Webpack's Hot Module Replacement (HMR). The idea is that instead of forcing a full refresh on modification, we patch the code that changed during the runtime.

The advantage of doing this is that it allows our application to retain its state. The process isn't fool proof, but when it works, it's quite neat. As a result we get good developer experience (DX).

You could achieve something similar by persisting your application state in other ways. For instance you could consider using `localStorage` for a similar purpose. You will still get a refresh, but it's far better than losing the entire state. You can reach the same result using multiple ways.

You can even implement the hot loading interface on your own. I'll show you the basic setup for a state container known as Redux[9]. It was designed with hot loading in mind and the approach works very well with it.

[8] https://babeljs.io/docs/usage/options/
[9] http://redux.js.org/

Setting Up *babel-preset-react-hmre*

A lot of the hard work has been done for us already. In order to configure our setup to support hot loading, we need to enable a Babel preset known as babel-preset-react-hmre[10] during development. To get started, install it:

```
npm i babel-preset-react-hmre --save-dev
```

Given it doesn't make sense to instrument our code with the hot loading logic for production usage, we should restrict it development only. One way to achieve this is to control *.babelrc* through `BABEL_ENV` environment variable.

If you are following the single file setup discussed in this book, we can control it using npm lifecycle event captured when npm is executed. This gives a predictable mapping between *package.json* and *.babelrc*. You can achieve this as follows:

webpack.config.js

```
...

const TARGET = process.env.npm_lifecycle_event;

...

process.env.BABEL_ENV = TARGET;

...
```

In addition we need to expand our Babel configuration to include the plugin we need during development. This is where that `BABEL_ENV` comes in. Babel determines the value of `env` like this:

1. Use the value of `BABEL_ENV` if set.
2. Use the value of `NODE_ENV` if set.
3. Default to `development`.

To connect `BABEL_ENV='start'` with Babel, configure as follows:

.babelrc

[10] https://www.npmjs.com/package/babel-preset-react-hmre

```
{
  "presets": [
    "es2015",
    "react"
  ],
  "env": {
    "start": {
      "presets": [
        "react-hmre"
      ]
    }
  }
}
```

After these steps your development setup should support hot loading. It is one of those features that makes development a little faster.

If you want to optimize your production build, consider studying Babel presets such as babel-preset-react-optimize[11].

If you prefer to see possible syntax errors at the browser console instead of hmre overlay, enable `webpack.NoErrorsPlugin()` at your Webpack `plugins` declaration.

Configuring Redux

In order to configure Redux reducers to support hot loading, we need to implement Webpack's hot loading protocol. Webpack provides a hook known as `module.hot.accept`. It gets called whenever Webpack detects a change. This allows you to reload and patch your code.

The idea is useful beyond Redux and can be used with other systems as well. To give you a rough implementation, consider the code below:

[11] https://www.npmjs.com/package/babel-preset-react-optimize

```
...

export default function configureStore(initialState) {
  const store = createStoreWithMiddleware(rootReducer, initialState);

  if(module.hot) {
    // Enable Webpack hot module replacement for reducers
    module.hot.accept('../reducers', () => {
      const nextReducer = require('../reducers/index').default;

      store.replaceReducer(nextReducer);
    });
  }

  return store;
}
```

The code doesn't do that much. It just waits for a change and then patches the code. The feasibility of patching depends on the underlying architecture. For a system like Redux it is simple given it was designed to be patched. It might be harder to pull off for something else.

You can find a full implementation of the idea online[12].

24.3 Using react-lite Instead of React for Production

React is quite heavy library even though the API is quite small considering. There are light alternatives, such as Preact[13] and react-lite[14]. react-lite implements React's API apart from features like `propTypes` and server side rendering. You lose out in debugging capabilities, but gain far smaller size. Preact implements a smaller subset of features and it's even smaller than react-lite.

Using react-lite in production instead of React can save around 100 kB minified code. Depending on your application, this can be a saving worth pursuing. Fortunately integrating react-lite is simple. It takes only a few lines of configuration to pull off.

To get started, install react-lite:

[12] https://github.com/survivejs-demos/redux-demo
[13] https://www.npmjs.com/package/preact
[14] https://www.npmjs.com/package/react-lite

```
npm i react-lite --save-dev
```

On the Webpack side, we can use a `resolve.alias` to point our React imports to react-lite instead:

```
resolve: {
  alias: {
    'react': 'react-lite',
    'react-dom': 'react-lite'
  }
}
```

If you try building your project with this setup, you should notice your bundle is considerably smaller.

A similar setup works for Preact too. In that case you would point to *preact-compat* instead. See preact-boilerplate[15] for the exact setup.

If you stick with vanilla React, you can still optimize it for production usage. See the *Setting Environment Variables* chapter to see how to achieve this.

24.4 Exposing React Performance Utilities to Browser

React provides a set of powerful performance related utilities[16] for figuring out how your application performs. Enabling them takes some setup. After the setup is done, you can access them through your browser console.

To get started, install the needed dependencies:

```
npm i expose-loader react-addons-perf --save-dev
```

Next we need to expose React to the console through the expose-loader[17]. The idea is that we'll bind the React performance utilities to that during development. Here's the Webpack loader configuration for exposing React as a global:

[15] https://github.com/developit/preact-boilerplate
[16] https://facebook.github.io/react/docs/perf.html
[17] https://www.npmjs.com/package/expose-loader

```
{
  test: require.resolve('react'),
  loader: 'expose?React'
}
```

After this you should be able to access `React` through a console. To make it possible to access the performance utilities, we need to do one more step. Add the following to the entry point of your application to enable `React.Perf` during development:

```
if(process.env.NODE_ENV !== 'production') {
  React.Perf = require('react-addons-perf');
}
```

If you check out the browser console now, you should be able to access the performance related API through `React.Perf`. The utilities allow you to understand better what's taking time and squeeze the last bits of performance out of your application. The *Elements* tab in Chrome can be useful as well. You can see how React operates on the DOM as it flashes.

It can be a good idea to install React Developer Tools[18] to Chrome for even more information. It allows you to inspect *props* and *state* of your application.

24.5 Optimizing Rebundling Speed During Development

We can optimize React's rebundling times during development by pointing the development setup to a minified version of React. The gotcha is that we will lose `propType` based validation! But if speed is more important, this technique may be worth a go. You can hide it behind an environment flag for instance if you want type checking.

In order to achieve what we want, we can use Webpack's `module.noParse` option. It accepts a RegExp or an array of RegExps. We can also pass full paths to it to keep our lives simple.

In addition to telling Webpack not to parse the minified file we want to use, we also need to point `react` to it. This can be achieved using a feature known as `resolve.alias` just like we did with *react-lite* above.

We can encapsulate the basic idea within a function like this:

[18] https://github.com/facebook/react-devtools

```
...

exports.dontParse = function(options) {
  const alias = {};
  alias[options.name] = options.path;

  return {
    module: {
      noParse: [
        options.path
      ]
    },
    resolve: {
      alias: alias
    }
  };
}
```

You would use the function like this assuming you are using webpack-merge[19]:

```
...

merge(
  common,
  dontParse({
    name: 'react',
    path: path.join(
      __dirname, 'node_modules', 'react', 'dist', 'react.min.js'
    )
  }),
  ...
)

...
```

If you try developing your application now, it should be at least a little bit faster to rebuild. The technique can be useful for production usage as well as you avoid some processing then.

`module.noParse` also accepts a regular expression. If we wanted to ignore all `*.min.js` files for instance, we could set it to `/\.min\.js/`. That can be a more generic way to solve the problem in some cases.

[19] https://www.npmjs.com/package/webpack-merge

Note that aliasing works also with loaders through resolveLoader.alias[20].

Not all modules support `module.noParse`, the files included by deps array should have no call to `require`, `define` or similar, or you will get an error when the app runs: `Uncaught ReferenceError: require is not defined`.

24.6 Setting Up Flow

Flow[21] performs static analysis based on your code and its type annotations. This means you will install it as a separate tool. You will then run it against your code. There's a Webpack plugin known as flow-status-webpack-plugin[22] that allows you to run it through Webpack during development.

When using React, the Babel preset does most of the work. It is able to strip Flow annotations and convert your code into a format that is possible to transpile further.

There's a Babel plugin known as babel-plugin-typecheck[23] that allows you to perform runtime checks based on your Flow annotations. After installing, you should just add the following section to the development section of your *.babelrc* to enable it:

.babelrc

```
"plugins": [
  [
    "typecheck"
  ]
]
```

Even though useful, Flow static checker is able to catch more errors. Runtime checks are still cool, though, and worth enabling if you are using Flow.

24.7 Setting Up TypeScript

Microsoft's TypeScript[24] is a far more established solution than Facebook's Flow. As a result you will find more premade type definitions for it and overall the quality of support should be better. You can use it with Webpack using at least the following loaders:

[20] https://webpack.github.io/docs/configuration.html#resolveloader
[21] http://flowtype.org/
[22] https://www.npmjs.com/package/flow-status-webpack-plugin
[23] https://www.npmjs.com/package/babel-plugin-typecheck
[24] http://www.typescriptlang.org/

- ts-loader[25]
- awesome-typescript-loader[26]
- typescript-loader[27]

24.8 Conclusion

There are a lot of aspects to keep in mind when configuring Webpack to work with React. Fortunately this is something you don't have to perform often. Once you have a solid basic setup fitting your needs together, it will take you far.

[25] https://www.npmjs.com/package/ts-loader
[26] https://www.npmjs.com/package/awesome-typescript-loader
[27] https://www.npmjs.com/package/typescript-loader