

SURVIVEJS

Webpack 5

FROM APPRENTICE TO MASTER



Juho Vepsäläinen

SurviveJS - Webpack 5

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 2022-10-26



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.



This work is licensed under a [Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License](#)

Contents

Foreword	i
Preface	ii
Introduction	iii
What is webpack	iii
How webpack changes the situation	iv
What will you learn	iv
How is the book organized	iv
Who is the book for	vi
What are the book conventions	vi
How is the book versioned	vii
How to get support	viii
Where to find additional material	viii
Acknowledgments	ix
What is Webpack	x
Webpack relies on modules	x
Webpack's execution process	xi
Webpack is configuration driven	xiv
Hot Module Replacement	xv
Asset hashing	xv
Code splitting	xvi
Webpack 5	xvi
Conclusion	xvii

I	Developing	1
1.	Getting Started	2
1.1	Setting up the project	2
1.2	Installing webpack	3
1.3	Running webpack	4
1.4	Setting up assets	5
1.5	Configuring mini-html-webpack-plugin	6
1.6	Examining the output	8
1.7	Adding a build shortcut	8
1.8	Conclusion	9
2.	Development Server	10
2.1	Webpack watch mode	10
2.2	webpack-dev-server	11
2.3	webpack-plugin-serve	12
2.4	Accessing development server from the network	15
2.5	Polling instead of watching files	15
2.6	Making it faster to develop webpack configuration	16
2.7	Watching files outside of webpack's module graph	17
2.8	Conclusion	17
3.	Composing Configuration	18
3.1	Possible ways to manage configuration	18
3.2	Composing configuration by merging	19
3.3	Setting up webpack-merge	20
3.4	Benefits of composing configuration	22
3.5	Configuration layouts	22
3.6	Conclusion	25
II	Styling	26
4.	Loading Styles	27
4.1	Loading CSS	27
4.2	Setting up initial CSS	29
4.3	PostCSS	29

CONTENTS

4.4	Using CSS preprocessors	30
4.5	Understanding css-loader lookups	30
4.6	Conclusion	32
5.	Separating CSS	33
5.1	Setting up MiniCssExtractPlugin	34
5.2	Managing styles outside of JavaScript	36
5.3	Conclusion	37
6.	Eliminating Unused CSS	38
6.1	Setting up Tailwind	38
6.2	Enabling PurgeCSS	41
6.3	Conclusion	44
7.	Autoprefixing	45
7.1	Setting up autoprefixing	45
7.2	Defining a browserslist	46
7.3	Conclusion	48
III	Loading Assets	49
8.	Loader Definitions	50
8.1	Anatomy of a loader	50
8.2	Loader evaluation order	51
8.3	Passing parameters to a loader	52
8.4	Inline definitions	53
8.5	Branching at use using a function	53
8.6	Loading with info object	54
8.7	Loading based on resourceQuery	55
8.8	Loading based on issuer	55
8.9	Alternate ways to match files	56
8.10	Understanding loader behavior	57
8.11	Conclusion	57
9.	Loading Images	58
9.1	Integrating images to the project	59
9.2	Using srcsets	60

CONTENTS

9.3	Optimizing images	60
9.4	Loading SVGs	61
9.5	Loading images dynamically	61
9.6	Loading sprites	62
9.7	Using placeholders	62
9.8	Referencing to images	63
9.9	Conclusion	64
10.	Loading Fonts	65
10.1	Setting up a loader	65
10.2	Using icon fonts	66
10.3	Using Google Fonts	66
10.4	Manipulating file-loader output path and <code>publicPath</code>	67
10.5	Eliminating unused characters	67
10.6	Generating font files based on SVGs	68
10.7	Conclusion	68
11.	Loading JavaScript	69
11.1	Using Babel with webpack configuration	70
11.2	Polyfilling features	74
11.3	Babel tips	75
11.4	Babel plugins	75
11.5	Generating differential builds	76
11.6	TypeScript	78
11.7	WebAssembly	80
11.8	Conclusion	80
IV	Building	81
12.	Source Maps	82
12.1	Inline source maps and separate source maps	83
12.2	Enabling source maps	83
12.3	Source map types supported by webpack	85
12.4	Inline source map types	85
12.5	Separate source map types	88
12.6	Other source map options	91

CONTENTS

12.7	SourceMapDevToolPlugin and EvalSourceMapDevToolPlugin	92
12.8	Changing source map prefix	92
12.9	Extracting source from source maps	92
12.10	Source maps on backend	93
12.11	Ignoring source map related warnings	93
12.12	Using dependency source maps	93
12.13	Conclusion	94
13.	Code Splitting	95
13.1	Code splitting formats	95
13.2	Controlling code splitting on runtime	99
13.3	Code splitting in React	100
13.4	Disabling code splitting	100
13.5	Machine learning driven prefetching	100
13.6	Conclusion	101
14.	Bundle Splitting	102
14.1	Adding something to split	102
14.2	Setting up a vendor bundle	103
14.3	Controlling bundle splitting	105
14.4	Splitting and merging chunks	106
14.5	Bundle splitting at entry configuration	108
14.6	Chunk types in webpack	108
14.7	Conclusion	109
15.	Tidying Up	110
15.1	Cleaning the build directory	110
15.2	Attaching a revision to the build	111
15.3	Copying files	113
15.4	Conclusion	113
V	Optimizing	114
16.	Minifying	115
16.1	Minifying JavaScript	115
16.2	Speeding up JavaScript execution	117

CONTENTS

16.3	Minifying HTML	117
16.4	Minifying CSS	118
16.5	Compressing bundles	120
16.6	Obfuscating output	120
16.7	Conclusion	120
17.	Tree Shaking	121
17.1	Demonstrating tree shaking	121
17.2	Tree shaking on package level	122
17.3	Tree shaking with external packages	123
17.4	Conclusion	123
18.	Environment Variables	124
18.1	The basic idea of DefinePlugin	125
18.2	Setting process.env.NODE_ENV	126
18.3	Choosing which module to use	127
18.4	Conclusion	128
19.	Adding Hashes to Filenames	129
19.1	Placeholders	129
19.2	Setting up hashing	131
19.3	Conclusion	133
20.	Separating a Runtime	134
20.1	Extracting a runtime	134
20.2	Using records	136
20.3	Integrating with asset pipelines	137
20.4	Conclusion	137
21.	Build Analysis	138
21.1	Configuring webpack	138
21.2	Enabling a performance budget	140
21.3	Dependency analysis	141
21.4	Composition analysis	142
21.5	Output plugins	144
21.6	Online services	144
21.7	Bundle comparison	145

CONTENTS

21.8	Unused files analysis	145
21.9	Duplication analysis	145
21.10	Understanding why a module was bundled	146
21.11	Conclusion	146
22.	Performance	147
22.1	Measuring impact	147
22.2	High-level optimizations	148
22.3	Low-level optimizations	148
22.4	Optimizing rebundling speed during development	149
22.5	Webpack 4 performance tricks	150
22.6	Conclusion	151
VI	Output	152
23.	Build Targets	153
23.1	Web targets	153
23.2	Node targets	154
23.3	Desktop targets	155
23.4	Conclusion	155
24.	Multiple Pages	156
24.1	Possible approaches	156
24.2	Generating multiple pages	157
24.3	Progressive web applications	159
24.4	Conclusion	160
25.	Server-Side Rendering	161
25.1	Setting up Babel with React	161
25.2	Setting up a React demo	162
25.3	Configuring webpack	163
25.4	Setting up a server	164
25.5	Open questions	165
25.6	Prerendering	166
25.7	Conclusion	167
26.	Module Federation	168

CONTENTS

26.1	Module federation example	169
26.2	Adding webpack configuration	170
26.3	Implementing the application with React	172
26.4	Separating bootstrap	173
26.5	Separating header	175
26.6	Pros and cons	179
26.7	Learn more	179
26.8	Conclusion	180

VII Techniques 181

27. Dynamic Loading	182
27.1 Dynamic loading with <code>require.context</code>	182
27.2 Dynamic paths with a dynamic <code>import</code>	183
27.3 Combining multiple <code>require.contexts</code>	184
27.4 Dealing with dynamic paths	185
27.5 Conclusion	185
28. Web Workers	186
28.1 Setting up a worker	186
28.2 Setting up a host	187
28.3 Sharing data	187
28.4 Other options	188
28.5 Conclusion	188
29. Internationalization	189
29.1 i18n with webpack	189
29.2 Setting up translations	190
29.3 Setting up webpack	190
29.4 Setting up application	192
29.5 Conclusion	193
30. Testing	194
30.1 Jest	194
30.2 Mocking	195
30.3 Removing files from tests	195

30.4	Conclusion	196
31.	Deploying Applications	197
31.1	Deploying with gh-pages	197
31.2	Deploying to other environments	199
31.3	Resolving <code>output.publicPath</code> dynamically	200
31.4	Conclusion	201
32.	Consuming Packages	202
32.1	<code>resolve.alias</code>	202
32.2	<code>resolve.modules</code>	203
32.3	<code>resolve.extensions</code>	203
32.4	<code>resolve.plugins</code>	203
32.5	Consuming packages outside of webpack	204
32.6	Dealing with globals	205
32.7	Managing symbolic links	206
32.8	Removing unused modules	207
32.9	Managing pre-built dependencies	208
32.10	Getting insights on packages	209
32.11	Conclusion	209
VIII	Extending	210
33.	Extending with Loaders	211
33.1	Debugging loaders with loader-runner	211
33.2	Implementing an asynchronous loader	213
33.3	Returning only output	214
33.4	Writing files	215
33.5	Passing options to loaders	217
33.6	Connecting custom loaders with webpack	219
33.7	Pitch loaders	220
33.8	Caching with loaders	222
33.9	Conclusion	223
34.	Extending with Plugins	224
34.1	The basic flow of webpack plugins	224

CONTENTS

34.2	Setting up a development environment	225
34.3	Implementing a basic plugin	227
34.4	Capturing options	228
34.5	Understanding compiler and compilation	229
34.6	Writing files through compilation	230
34.7	Managing warnings and errors	232
34.8	Plugins can have plugins	232
34.9	Conclusion	233

Conclusion	234
General checklist	234
Development checklist	234
Production checklist	235
Conclusion	236

Appendices 237

Comparison of Build Tools	238
Task runners	238
Script loaders	243
Bundlers	245
Zero configuration bundlers	248
Other Options	248
Conclusion	249

Hot Module Replacement	251
Enabling HMR	251
Implementing the HMR interface	254
Setting WDS entry points manually	256
HMR and dynamic loading	257
Conclusion	257

CSS Modules	258
CSS Modules through css-loader	258
Using CSS Modules with third-party libraries and CSS	260

CONTENTS

Conclusion	260
Searching with React	261
Implementing search with code splitting	261
Conclusion	264
Troubleshooting	265
Module related errors	266
DeprecationWarning	267
Conclusion	268
Glossary	269

Foreword

It's a funny story how I started with webpack. Before getting addicted to JavaScript, I also developed in Java. I tried GWT (Google Web Toolkit) in that time. GWT is a Java-to-JavaScript Compiler, which has a great feature: [code splitting](#)¹. I liked this feature and missed it in existing JavaScript tooling. I opened [an issue](#)² to an existing module bundler, but it did not go forward. Webpack was born.

Somehow the Instagram frontend team discovered an early version of webpack and started to use it for [instagram.com](#). Pete Hunt, Facebook employee managing the Instagram web team, gave [the first significant talk about webpack](#)³ at OSCON 2014. The talk boosted the popularity of webpack. One of the reasons for adoption of webpack by Instagram was **code splitting**.

I have been following this book since its early stages. It was once a combined React and webpack book. It has grown since then and become a book of its own filled with content.

Juho is an important part of the webpack documentation team for the webpack documentation, so he knows best what complements the official documentation. He has used this knowledge to create a book that supplies you with a deep understanding of webpack and teaches you to use the tool to its full potential.

Tobias Koppers

¹<http://www.gwtproject.org/doc/latest/DevGuideCodeSplitting.html>

²<https://github.com/medikoo/modules-webmake/issues/7>

³<https://www.youtube.com/watch?v=VktCL6Nqm6Y>

Preface

The book you are reading right now goes years back. It all started with a comment I made on Christian Alfoni's blog in 2014. It was when I discovered webpack and React, and I felt there was a need for a cookbook about the topics. The work began with a GitHub wiki in early 2015.

After a while, I realized this should become an *actual* book and tried pitching it to a known publisher. As they weren't interested yet and I felt the book needed to happen, I started the **SurviveJS** effort. The book warped into "SurviveJS - Webpack and React", my first self-published book. It combined the two topics into one single book.

Because a book focusing only on a single technology can stand taller, I split the book into two separate ones in 2016. The current edition represents the webpack portion of it, and it has grown significantly due to this greater focus. The journey has not been a short one, but it has been possible thanks to community support and continued interest in the topic.

During these years, webpack has transformed. Instead of relying on a single prolific author, a core team has grown around the project and has attracted more people to support the effort.

There is an [open collective campaign](https://opencollective.com/webpack)⁴ to help the project to succeed financially. I am donating 30% of the book earnings to Tobias to support the project. By supporting the book, you help webpack development as well.

Juho Vepsäläinen

⁴<https://opencollective.com/webpack>

Introduction

[Webpack](https://webpack.js.org/)⁵ simplifies web development by solving a fundamental problem: bundling. It takes in various assets, such as JavaScript, CSS, and HTML, and transforms them into a format that's convenient to consume through a browser. Doing this well takes a significant amount of pain away from web development.

It's not the most accessible tool to learn due to its configuration-driven approach, but it's incredibly powerful. The purpose of this guide is to help you get started with webpack and go beyond the basics.

What is webpack

Web browsers consume HTML, CSS, JavaScript, and multimedia files. As a project grows, tracking all of these files and adapting them to different targets (e.g. browsers) becomes too complicated to manage without help. Webpack addresses these problems. Managing complexity is one of the fundamental issues of web development, and solving this problem well helps significantly.

Webpack isn't the only available bundler, and a collection of different tools have emerged. Task runners, such as Grunt and Gulp, are good examples of higher-level tools. Often the problem is that you need to write the workflows by hand. Pushing that issue to a bundler, such as webpack, is a step forward.

Framework specific abstractions, such as [create-react-app](https://create-react-app.dev/)⁶, [rockpack](https://www.rockpack.io/)⁷, or [@angular/cli](https://www.npmjs.com/package/@angular/cli)⁸, use webpack underneath. That said, there's still value in understanding the tool if you have to customize the setup.



[Turbopack](https://turbo.build/pack)⁹ is the official successor to webpack. It has been rewritten in Rust with performance in mind out of the box.

⁵<https://webpack.js.org/>

⁶<https://create-react-app.dev/>

⁷<https://www.rockpack.io/>

⁸<https://www.npmjs.com/package/@angular/cli>

⁹<https://turbo.build/pack>

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 the standard `require` or `import` statements. You can refer to your style files and other assets the same way if you want.

Webpack does all the preprocessing for you and gives you the bundles you specify through its configuration and code. This declarative approach is versatile, but it's challenging to learn.

Webpack becomes an indispensable tool after you begin to understand how it works. This book exists to get through that initial learning curve and even go further.

What will you learn

This book has been written to complement the official documentation of webpack and it helps you to get through the initial learning curve and go further.

You will learn to develop a composable webpack configuration for both development and production purposes. Advanced techniques covered by the book allow you to get the most out of webpack.

How is the book organized

The book starts by explaining what webpack is. After that, you will find multiple chapters that discuss webpack from a different viewpoint. As you go through these chapters, you will develop your webpack configuration while at the same time learning essential techniques.

The book consists of the following parts:

- **Developing** gets you up and running with webpack. This part goes through features such as automatic browser refresh and explains how to compose your configuration so that it remains maintainable.

- **Styling** puts heavy emphasis on styling related topics. You will learn how to load styles with webpack and introduce techniques such as autoprefixing into your setup.
- **Loading** explains webpack's loader definitions in detail and shows you how to load assets such as images, fonts, and JavaScript.
- **Building** introduces source maps and the ideas of bundle and code splitting. You will learn to tidy up your build.
- **Optimizing** pushes your build to production quality level and introduces many smaller tweaks to make it smaller. You will learn to tune webpack for performance.
- **Output** discusses webpack's output related techniques. Despite its name, it's not only for the web. You see how to manage multiple page setups with webpack, pick up the basic idea of Server-Side Rendering, and learn about Module Federation.
- **Techniques** discusses several specific ideas, including dynamic loading, web workers, internationalization, deploying your applications, and consuming npm packages through webpack.
- **Extending** shows how to extend webpack with loaders and plugins.

Finally, there is a short conclusion chapter that recaps the main points of the book. It contains checklists of techniques from this book that allow you to go through your projects methodically.

The appendices at the end of the book cover secondary topics and sometimes dig deeper into the main ones. You can approach them in any order you want, depending on your interest.

The *Troubleshooting* appendix at the end covers what to do when webpack gives you an error. It includes a process, so you know what to do and how to debug the problem. When in doubt, study the appendix. If you are unsure of a term and its meaning, see the *Glossary* at the end of the book.

Who is the book for

The book has been written mainly beginner and intermediate developers in mind. For experts that already know webpack well, there's value in the form of techniques. The book summaries included in each chapter and at the *Conclusion* chapter, make it fast to skim and pick up the ideas.

Especially at the beginning and intermediate levels it can make sense to follow the book tutorial and develop your own webpack configuration from scratch and then check the chapters that feel most relevant to you. The only expectation is that you have a basic knowledge of JavaScript, Node, and npm.

Even if you use webpack through an abstraction such as Create React App, it can be valuable to understand the tool in case you have to extend your setup one day. Many of the techniques discussed go beyond webpack itself and are useful to know in daily development if and when you have to optimize your web application or site for example.

What are the book conventions

The book uses several conventions to keep the content accessible. I've listed examples below:



This is a tip. Often you can find auxiliary information and further references in tips.



This is a warning that's highlighting unexpected behavior or a common problem point that you should know.

Especially in the early part of the book, the code is written in a tutorial form. For this reason, the following syntax is used:

```
// You might see insertions
const webpack = require("webpack");

// You might see deletions as well
const { MiniHtmlWebpackPlugin } = require("mini-html-webpack-plugin");

// Or combinations of both
const { MiniHtmlWebpackPlugin } = require("mini-html-webpack-plugin");
const webpack = require("webpack");

// If content has been omitted, then ellipsis is used
...
```

Sometimes the code assumes addition without the highlighting for insertion and many examples of the book work without by themselves and I've crosslinked to prerequisites where possible.

You'll also see code within sentences and occasionally important terms have been **highlighted**. You can find the definition of these terms at the *Glossary*.

How is the book versioned

The book uses a versioning scheme, and release notes for each new version are maintained at [the book blog](https://survivejs.com/blog/)¹⁰. You can also use GitHub *compare* tool for this purpose. Example:

<https://github.com/survivejs/webpack-book/compare/v3.0.0...v3.0.11>

The page shows 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.

The current version of the book is **3.0.11**.

¹⁰<https://survivejs.com/blog/>

How to get support

If you run into trouble or have questions related to the content, there are several options:

- Contact me through [GitHub Issue Tracker](#)¹¹.
- Join me at [Gitter Chat](#)¹².
- Send me an email at info@survivejs.com¹³.
- Ask me anything about webpack at [SurviveJS AmA](#)¹⁴.

If you post questions to Stack Overflow, tag them using **survivejs**. You can use the hashtag **#survivejs** on Twitter for the same result.

I am available for commercial consulting. In my past work, I have helped companies to optimize their usage of webpack. The work has an impact on both developer experience and the end-users in the form of a more performant and optimized build.

Where to find additional material

You can find more related material from the following sources:

- Join the [mailing list](#)¹⁵ for occasional updates.
- Follow [@survivejs](#)¹⁶ on Twitter.
- Subscribe to the [blog RSS](#)¹⁷ to get access to interviews and more.
- Subscribe to the [Youtube channel](#)¹⁸.
- Check out [SurviveJS related presentation slides](#)¹⁹.

¹¹<https://github.com/survivejs/webpack-book/issues>

¹²<https://gitter.im/survivejs/webpack>

¹³<mailto:info@survivejs.com>

¹⁴<https://github.com/survivejs/ama/issues>

¹⁵<https://buttondown.email/SurviveJS>

¹⁶<https://twitter.com/survivejs>

¹⁷<https://survivejs.com/atom.xml>

¹⁸<https://www.youtube.com/SurviveJS>

¹⁹<https://presentations.survivejs.com/>

Acknowledgments

Big thanks to [Christian Alfoni](http://www.christianalfoni.com/)²⁰ for helping me craft the first version of this book as this inspired the entire SurviveJS effort. The text you see now is a complete rewrite.

This book wouldn't be half as good as it is without patient editing and feedback by my editors [Jesús Rodríguez](https://github.com/Foxandxss)²¹, [Artem Sapegin](https://github.com/sapegin)²², and [Pedr Browne](https://github.com/Undistracted)²³. Thank you.

This book wouldn't have been possible without the original “SurviveJS - Webpack and React” effort. Anyone who contributed to it deserves my thanks. You can check that book for more accurate 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, Jake Goulding, Andrew Ferk, gabo, Giang Nguyen, @Coaxial, @khronic, Henrik Raitasola, Gavin Orland, David Riccitelli, Stephen Wright, Majky Bašista, Gunnari Auvinen, Jón Levy, Alexander Zaytsev, Richard Muller, Ava Mallory (Fiverr), Sun Zheng'an, Nancy (Fiverr), Aluan Haddad, Steve Mao, Craig McKenna, Tobias Koppers, Stefan Frede, Vladimir Grenaderov, Scott Thompson, Rafael De Leon, Gil Forcada Codinachs, Jason Aller, @pikeshawn, Stephan Klinger, Daniel Carral, Nick Yianilos, Stephen Bolton, Felipe Reis, Rodolfo Rodriguez, Vicky Koblinski, Pyotr Ermishkin, Ken Gregory, Dmitry Kaminski, John Darryl Peling, Brian Cui, @st-sloth, Nathan Klatt, Muhamadamin Ibragimov, Kema Akpala, Roberto Fuentes, Eric Johnson, Luca Poldelmengo, Giovanni Iembo, Dmitry Anderson, Douglas Cerna, Chris Blossom, Bill Fienberg, Andrey Bushman, Andrew Staroscik, Cezar Neaga, Eric Hill, Jay Somedon, Luca Fagioli, @cdoublev, Boas Mollig, Shahin Sheidaei, Stefan Frede, Dennis Weiershäuser, Tommy-Pepsi Gaudreau, Andrea Maschio, Kusal KC, PrinceRajRoy, Cody Casey, Kahlil Hodgson, Fahad Amin Shovon, Justin Wen, Rajiv Seelam, Steve Higham, and many others who have contributed direct feedback for this book!

²⁰<http://www.christianalfoni.com/>

²¹<https://github.com/Foxandxss>

²²<https://github.com/sapegin>

²³<https://github.com/Undistracted>

What is Webpack

Webpack is a **module bundler**. Webpack can take care of bundling alongside a separate task runner. However, the line between bundler and task runner has become blurred thanks to community-developed webpack plugins. Sometimes these plugins are used to perform tasks that are usually done outside of webpack, such as cleaning the build directory or deploying the build although you can defer these tasks outside of webpack.

React, and **Hot Module Replacement** (HMR) helped to popularize webpack and led to its usage in other environments, such as [Ruby on Rails](#)²⁴. Despite its name, webpack is not limited to the web alone. It can bundle with other targets as well, as discussed in the *Build Targets* chapter.



If you want to understand build tools and their history in better detail, check out the *Comparison of Build Tools* appendix.

Webpack relies on modules

The smallest project you can bundle with webpack consists of **input** and **output**. The bundling process begins from user-defined **entries**. Entries themselves are **modules** and can point to other modules through **imports**.

When you bundle a project using webpack, it traverses the imports, constructing a **dependency graph** of the project and then generates **output** based on the configuration. Additionally, it's possible to define **split points** to create separate bundles within the project code itself.

Internally webpack manages the bundling process using what's called **chunks** and the term often comes up in webpack related documentation. Chunks are smaller pieces of code that are included in the bundles seen in webpack output.

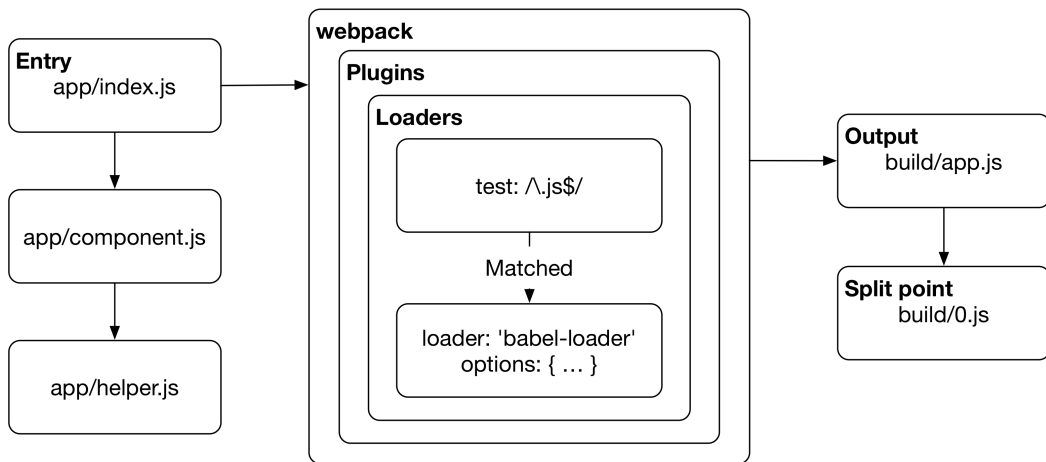
²⁴<https://github.com/rails/webpacker>

Webpack supports ES2015, CommonJS, MJS, and AMD module formats out of the box. There's also support for [WebAssembly²⁵](#), a new way of running low-level code in the browser. The loader mechanism works for CSS as well, with `@import` and `url()` support through **css-loader**. You can find plugins for specific tasks, such as minification, internationalization, HMR, and so on.



A dependency graph is a directed graph that describes how nodes relate to each other. In this case, the graph definition is defined through references (`require`, `import`) between files. Webpack statically traverses these without executing the source to generate the graph it needs to create bundles.

Webpack's execution process



Webpack's execution process

Webpack begins its work from **entries**. Often these are JavaScript modules where webpack begins its traversal process. During this process, webpack evaluates entry matches against **loader** configurations that tell webpack how to transform each match.

²⁵<https://developer.mozilla.org/en-US/docs/WebAssembly>



Starting from webpack 5, there's support for [experiments²⁶](#). These represent future functionality that's hidden behind a feature flag and allows early testing.

Resolution process

An entry itself is a module and when webpack encounters one, it tries to match the module against the file system using the `resolve` configuration. For example, you can tell webpack to perform the lookup against specific directories in addition to `node_modules`.



It's possible to adjust the way webpack matches against file extensions, and you can define specific aliases for directories. The *Consuming Packages* chapter covers these ideas in greater detail.

If the resolution pass failed, webpack will raise a runtime error. If webpack managed to resolve a file, webpack performs processing over the matched file based on the loader definition. Each loader applies a specific transformation against the module contents.

The way a loader gets matched against a resolved file can be configured in multiple ways, including by file type and by location within the file system. Webpack's flexibility even allows you to apply a specific transformation to a file based on *where* it was imported into the project.

The same resolution process is performed against webpack's loaders. Webpack allows you to apply similar logic when determining which loader it should use. Loaders have `resolve` configurations of their own for this reason. If webpack fails to perform a loader lookup, it will raise a runtime error.



To resolve, webpack relies on [enhanced-resolve²⁷](#) package underneath.

²⁶<https://webpack.js.org/configuration/experiments/#experiments>

²⁷<https://www.npmjs.com/package/enhanced-resolve>

Webpack resolves against any file type

Webpack will resolve each module it encounters while constructing the dependency graph. If an entry contains dependencies, the process will be performed recursively against each dependency until the traversal has completed. Webpack can perform this process against any file type, unlike specialized tools like the Babel or Sass compiler.

Webpack gives you control over how to treat different assets it encounters. For example, you can decide to **inline** assets to your JavaScript bundles to avoid requests. Webpack also allows you to use techniques like CSS Modules to couple styling with components. Webpack ecosystem is filled with plugins that extend its capabilities.

Although webpack is used mainly to bundle JavaScript, it can capture assets like images or fonts and emit separate files for them. Entries are only a starting point of the bundling process and what webpack emits depends entirely on the way you configure it.

Evaluation process

Assuming all loaders were found, webpack evaluates the matched loaders from bottom to top and right to left (`styleLoader(cssLoader('./main.css'))`) while running the module through each loader in turn. As a result, you get output which webpack will inject in the resulting **bundle**. The *Loader Definitions* chapter covers the topic in detail.

If loader evaluation completed without a runtime error, webpack includes the source in the bundle. Although loaders can do a lot, they don't provide enough power for advanced tasks. Plugins can intercept **runtime events** supplied by webpack.

A good example is bundle extraction performed by the `MiniCssExtractPlugin` which, when used with a loader, extracts CSS files out of the bundle and into a separate file. Without this step, CSS would be inlined in the resulting JavaScript, as webpack treats all code as JavaScript by default. The extraction idea is discussed in the *Separating CSS* chapter.

Finishing

After every module has been evaluated, webpack writes **output**. The output is a small runtime that executes the result in a browser and a manifest listing bundles to load. The runtime can be extracted to a file of its own, as discussed later in the book.

That's not all there is to the bundling process. For example, you can define specific **split points** where webpack generates separate bundles that are loaded based on application logic. This idea is discussed in the *Code Splitting* chapter.

Webpack is configuration driven

At its core, webpack relies on configuration as in the sample below:

webpack.config.js

```
const path = require("path");
const webpack = require("webpack");

module.exports = {
  entry: { app: "./entry.js" }, // Start bundling
  output: {
    path: path.join(__dirname, "dist"), // Output to dist directory
    filename: "[name].js", // Emit app.js by capturing entry name
  },
  // Resolve encountered imports
  module: {
    rules: [
      { test: /\.css$/, use: ["style-loader", "css-loader"] },
      { test: /\.js$/, use: "swc-loader", exclude: /node_modules/ },
    ],
  },
  // Perform additional processing
  plugins: [new webpack.DefinePlugin({ HELLO: "hello" })],
  // Adjust module resolution algorithm
  resolve: { alias: { react: "preact-compat" } },
};
```

Webpack's configuration model can feel a bit opaque at times as the configuration file can appear monolithic and it can be difficult to understand what webpack is doing unless you know the ideas behind it. The book exists to make the concepts and ideas to address this problem.



Often webpack's property definitions are flexible and it's the best to look at either the documentation or TypeScript definitions to see what's allowed. For example, `entry` can be a function and an asynchronous one even. At times, there are multiple ways to achieve the same, especially with loaders.



Webpack's plugins are registered from top to bottom but loaders follow the opposite rule. That means if you add a loader definition after the existing ones and it matches the same test, it will be evaluated first. See the *Loader Definitions* chapter to understand the different possibilities better.

Hot Module Replacement

You are likely familiar with tools, such as [LiveReload](http://livereload.com/)²⁸ or [BrowserSync](http://www.browsersync.io/)²⁹, already. These tools refresh the browser automatically as you make changes. *Hot Module Replacement* (HMR) takes things one step further. In the case of React, it allows the application to maintain its state without forcing a refresh. While this does not sound all that special, it can make a big difference in practice.

Asset hashing

With webpack, you can inject a hash to each bundle name (e.g., `app.d587bbd6.js`) 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.

²⁸<http://livereload.com/>

²⁹<http://www.browsersync.io/>

Code splitting

In addition to HMR, webpack's bundling capabilities are extensive. Webpack allows you to split code in various ways. You can even load code dynamically as your application gets executed. This sort of lazy loading comes in handy, especially for broader applications, as dependencies can be loaded on the fly as needed.

Even small applications can benefit from **code splitting**, as it allows the users to get something usable in their hands faster. Performance is a feature, after all. Knowing the basic techniques is worthwhile.

Webpack 5

Webpack 5 is a new version of the tool that promises the following changes:

- There's better caching behavior during development - now it reuses disk-based cache between separate runs.
- Micro frontend style development is supported through *Module Federation* and you can learn more about it in the chapter.
- Internal APIs (esp. plugins) have been improved and older APIs have been deprecated.
- The development and production targets have better defaults. For example now `contenthash` is used for production resulting in predictable caching behavior. The topic is discussed in detail at the *Adding Hashes to Filename* chapter.

[Webpack 5 release post](https://webpack.js.org/blog/2020-10-10-webpack-5-release/)³⁰ lists all the major changes. Apart from the caching improvements and *Module Federation*, it can be considered a clean up release.

There's [an official migration guide](https://webpack.js.org/migrate/5/)³¹ that lists all of the changes that have to be done to port a project from webpack 4 to 5.

It's possible that a project will run without any changes to the configuration but that you'll receive deprecation warnings. To find out where they are coming from, use `node --trace-deprecation node_modules/webpack/bin/webpack.js` when running webpack.

³⁰<https://webpack.js.org/blog/2020-10-10-webpack-5-release/>

³¹<https://webpack.js.org/migrate/5/>

Conclusion

Webpack comes with a significant learning curve. However, it's a tool worth learning, given how much time and effort it can save over the long term. To get a better idea how it compares to others, check out the *Comparison of Build Tools* appendix.

Webpack won't solve everything. However, it does solve the problem of bundling. That's one less worry during development.

To summarize:

- Webpack is a **module bundler**, but you can also use it running tasks as well.
- Webpack relies on a **dependency graph** underneath. Webpack traverses through the source to construct the graph, and it uses this information and configuration to generate bundles.
- Webpack relies on **loaders** and **plugins**. Loaders operate on a module level, while plugins rely on hooks provided by webpack and have the best access to its execution process.
- Webpack's **configuration** describes how to transform assets of the graphs and what kind of output it should generate. Part of this information can be included in the source itself if features like **code splitting** are used.
- **Hot Module Replacement** (HMR) helped to popularize webpack. It's a feature that can enhance the development experience by updating code in the browser without needing a full page refresh.
- Webpack can generate **hashes** for filenames allowing you to invalidate past bundles as their contents change.

In the next part of the book, you'll learn to construct a development configuration using webpack while learning more about its basic concepts.

I Developing

In this part, you get up and running with webpack. You will learn to configure **webpack-plugin-serve**. Finally, you compose the configuration so that it's possible to expand in the following parts of the book.

1. Getting Started

Before getting started, make sure you are using a recent version of [Node](http://nodejs.org/)¹. You should use at least the most current LTS (long-term support) version as the configuration of the book has been written with modern Node features in mind.

You should have `node` and `npm` (or `yarn`) commands available at your terminal. To get a more controlled environment, use [Docker](https://www.docker.com/)², [nvm](https://www.npmjs.com/package/nvm)³, or a similar tool.



The completed configuration is available at [GitHub](https://github.com/survivejs-demos/webpack-demo)⁴ for reference.

1.1 Setting up the project

To get a starting point, create a directory for the project, and set up a `package.json` there as `npm` uses that to manage project dependencies.

```
mkdir webpack-demo
cd webpack-demo
# -y generates a `package.json` with default values
# Set the defaults at ~/.npmrc
npm init -y
```

You can tweak the generated `package.json` manually to make further changes to it even though a part of the operations modify the file automatically for you. The official documentation explains [package.json options](https://docs.npmjs.com/files/package.json)⁵ in more detail.

¹<http://nodejs.org/>

²<https://www.docker.com/>

³<https://www.npmjs.com/package/nvm>

⁴<https://github.com/survivejs-demos/webpack-demo>

⁵<https://docs.npmjs.com/files/package.json>



This is an excellent chance to set up version control using [Git](#)⁶. You can create a commit per step and tag per chapter, so it's easier to move back and forth if you want.

1.2 Installing webpack

Even though webpack can be installed globally (`npm add webpack -g`), it's a good idea to maintain it as a dependency of your project to avoid issues, as then you have control over the exact version you are running.

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

To add webpack to the project, run:

```
npm add webpack webpack-nano -D # --save-dev
```

You should see **webpack** and **webpack-nano** in 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 in the `node_modules/.bin` directory.

We're using [webpack-nano](#)⁷ over the official [webpack-cli](#)⁸ as it has enough features for the book project while being directly compatible with webpack 4 and 5.

webpack-cli comes with additional functionality, including `init` and `migrate` commands that allow you to create new webpack configuration fast and update from an older version to a newer one.



`npm add` is an alias for `npm install`. It's used in the book as it aligns well with Yarn and `yarn add`. You can use which one you prefer.

⁶<https://git-scm.com/>

⁷<https://www.npmjs.com/package/webpack-nano>

⁸<https://www.npmjs.com/package/webpack-cli>

1.3 Running webpack

Type `node_modules/.bin/wp` to run the locally installed **webpack-nano**.

After running, you should see a version, a link to the command line interface guide and an extensive list of options. Most aren't used in this project, but it's good to know that this tool is packed with functionality if nothing else.

```
$ node_modules/.bin/wp
[ ] webpack: Build Finished
[ ] webpack: assets by status 0 bytes [cached] 1 asset

WARNING in configuration
  The 'mode' option has not been set, webpack will fallback to 'production' for this value. Set 'mode' option to 'development' or 'production' to enable defaults for each environment.
  You can also set it to 'none' to disable any default behavior. Learn more: https://webpack.js.org/configuration/mode/

ERROR in main
Module not found: Error: Can't resolve './src' in 'webpack-demo'

webpack 5.5.0 compiled with 1 error and 1 warning in 115 ms
```

The output tells that webpack cannot find the source to compile. Ideally we would pass mode parameter to it as well to define which defaults we want.

To make webpack compile, do the following:

1. Set up `src/index.js` with something like `console.log("Hello world");`.
2. Execute `node_modules/.bin/wp`. Webpack will discover the source file by convention.
3. Examine `dist/main.js`. You should see webpack bootstrap code that begins executing the code. Below the bootstrap, you should find something familiar.



You can display the exact path of the executables using `npm bin`. Most likely it points to `./node_modules/.bin`.

1.4 Setting up assets

To make the build more complex, we can add another module to the project and start developing a small application:

src/component.js

```
export default (text = "Hello world") => {  
  const element = document.createElement("div");  
  element.innerHTML = text;  
  return element;  
};
```

We also have to modify the original file to import the new file and render the application to the DOM:

src/index.js

```
import component from "../component";  
  
document.body.appendChild(component());
```

Examine the output after building the project by running `node_modules/.bin/wp` again. You should see both modules in the bundle that webpack wrote to the `dist` directory. One problem remains, though. How can we test the application in the browser?

1.5 Configuring mini-html-webpack-plugin

The problem can be solved by writing an `index.html` file that points to the generated file. Instead of doing that on our own, we can use a webpack plugin to do this.

To get started, install [mini-html-webpack-plugin](https://www.npmjs.com/package/mini-html-webpack-plugin)⁹:

```
npm add mini-html-webpack-plugin -D
```



[html-webpack-plugin](https://www.npmjs.com/package/html-webpack-plugin)¹⁰ is a versatile option that can be expanded with plugins. For anything beyond basic usage, it's a good option.

To connect the plugin with webpack, set up the configuration as below:

webpack.config.js

```
const { mode } = require("webpack-nano/argv");
const {
  MiniHtmlWebpackPlugin,
} = require("mini-html-webpack-plugin");

module.exports = {
  mode,
  plugins: [
    new MiniHtmlWebpackPlugin({ context: { title: "Demo" } }),
  ],
};
```

⁹<https://www.npmjs.com/package/mini-html-webpack-plugin>

¹⁰<https://www.npmjs.com/package/html-webpack-plugin>

Now that the configuration is done, try the following:

1. Build the project using `node_modules/.bin/wp --mode production`. You can try the `development` and `none` modes too.
2. Run a static file server using `npx serve dist` or a similar command you are familiar with.



The `none` mode doesn't apply any defaults. Use it for debugging.



`npx` is installed with `npm` and could be used to run `npm` packages without installation, as well as to run locally installed packages.

You should see a hello message in your browser:

Hello world

Hello world



In addition to a configuration object, webpack accepts an array of configurations. You can also return a `Promise` that eventually resolves to a configuration. Latter is useful if you are fetching configuration related data from a third-party source.



Webpack has default configuration for its entries and output. It looks for source from `./src` by default and it emits output to `./dist`. You can control these through `entry` and `output` respectively as seen in the *What is Webpack* chapter.

1.6 Examining the output

If you execute `node_modules/.bin/wp --mode production`, you should see output:

```
webpack: Build Finished
webpack: asset index.html 198 bytes [compared for emit]
      asset main.js 136 bytes [compared for emit] [minimized] (name: main)
      orphan modules 140 bytes [orphan] 1 module
      ./src/index.js + 1 modules 217 bytes [built] [code generated]
      webpack 5.5.0 compiled successfully in 193 ms
```

Starting from webpack 5, the output has been simplified and it's largely self-explanatory. The default output has improved as well as you can see by studying `dist/main.js`. Earlier it contained an entire webpack runtime but starting from webpack 5, the tool is able to optimize the result to a minimum required.

1.7 Adding a build shortcut

Given executing `node_modules/.bin/wp --mode production` gets boring after a while, let's adjust `package.json` to run tasks as below:

package.json

```
{
  "scripts": {
    "build": "wp --mode production"
  }
}
```

Run `npm run build` to see the same output as before. `npm` adds `node_modules/.bin` temporarily to the path enabling this. As a result, rather than having to write `"build": "node_modules/.bin/wp"`, you can do `"build": "wp"`.

You can execute this kind of scripts through `npm run` and you can use the command anywhere within your project. If you run the command without any parameters (`npm run`), it gives you the listing of available scripts.



If you want to run multiple commands concurrently, see the [concurrently](https://www.npmjs.com/package/concurrently)¹¹ package. It has been designed to allow that while providing neat output.

1.8 Conclusion

Even though you have managed to get webpack up and running, it does not do that much yet. Developing against it would be painful as we would have to recompile all the time. That's where webpack's more advanced features we explore in the next chapters come in.

To recap:

- It's a good idea to use a locally installed version of webpack over a globally installed one. This way you can be sure of what version you are using. The local dependency also works in a Continuous Integration environment.
- Webpack provides a command line interface through the **webpack-cli** package. You can use it even without configuration, but any advanced usage requires a config file. **webpack-nano** is a good alternative for basic usage.
- To write more complicated setups, you most likely have to write a separate `webpack.config.js` file.
- **mini-html-webpack-plugin** and **html-webpack-plugin** can be used to generate an HTML entry point to your application. In the *Multiple Pages* chapter you will see how to generate multiple separate pages using these plugins.
- It's handy to use npm scripts in `package.json` to manage webpack. You can use them as a light task runner and use system features outside of webpack.

In the next chapter, you will learn how to improve the developer experience by enabling automatic browser refresh.

¹¹<https://www.npmjs.com/package/concurrently>

2. Development Server

When developing a frontend without any special tooling, you often end up having to refresh the browser to see changes. Given this gets annoying fast, there's tooling to remedy the problem.

The first tools on the market were [LiveReload](http://livereload.com/)¹ and [Browsersync](http://www.browsersync.io/)². The point of either is to allow refreshing the browser automatically as you develop. They also pick up CSS changes and apply the new style without a hard refresh that loses the state of the browser.

It's possible to setup Browsersync to work with webpack through [browser-sync-webpack-plugin](https://www.npmjs.com/package/browser-sync-webpack-plugin)³, but webpack has more tricks in store in the form of a watch mode, and a development server.

2.1 Webpack watch mode

Webpack's watch mode rebuilds the bundle on any change of the project files. It can be activated either by setting watch field true in webpack configuration or by passing the `--watch` to webpack-cli.

Although this solves the problem of recompiling your source on change, it does nothing on the frontend side and browser updates. That's where further solutions are required.

¹<http://livereload.com/>

²<http://www.browsersync.io/>

³<https://www.npmjs.com/package/browser-sync-webpack-plugin>

2.2 webpack-dev-server

[webpack-dev-server](https://www.npmjs.com/package/webpack-dev-server)⁴ (WDS) is the officially maintained development server running **in-memory**, meaning the bundle contents aren't written out to files but stored in memory. The distinction is vital when trying to debug code and styles.

If you go with WDS, there are a couple of relevant fields that you should be aware of:

- `devServer.historyApiFallback` should be set if you rely on HTML5 History API based routing.
- `devServer.contentBase` - Assuming you don't generate `index.html` dynamically and prefer to maintain it yourself in a specific directory, you need to point WDS to it. `contentBase` accepts either a path (e.g., `"build"`) or an array of paths (e.g., `["build", "images"]`). The value defaults to the project root.
- `devServer.proxy` - If you are using multiple servers, you have to proxy WDS to them. The proxy setting accepts an object of proxy mappings (e.g., `{ "/api": "http://localhost:3000/api" }`) that resolve matching queries to another server. Proxying is disabled by default.
- `devServer.headers` - Attach custom headers to your requests here.



To integrate with another server, it's possible to emit files from WDS to the file system by setting `devServer.writeToDisk` property to `true`.



You should use WDS strictly for development. If you want to host your application, consider other solutions, such as Apache or Nginx.



WDS depends implicitly on **webpack-cli** in command line usage.

⁴<https://www.npmjs.com/package/webpack-dev-server>

2.3 webpack-plugin-serve

[webpack-plugin-serve](https://www.npmjs.com/package/webpack-plugin-serve)⁵ (WPS) is a third-party plugin that wraps the logic required to update the browser into a webpack plugin. Underneath it relies on webpack's watch mode, and it builds on top of that while implementing **Hot Module Replacement** (HMR) and other features seen in WDS.

WPS also supports webpack's multi-compiler mode (i.e., when you give an array of configurations to it) and a status overlay.

Given webpack's watch mode emits to the file system by default, WPS provides an option for [webpack-plugin-ramdisk](https://www.npmjs.com/package/webpack-plugin-ramdisk)⁶ to write to the RAM instead. Using the option improves performance while avoiding excessive writes to the file system.

Getting started with webpack-plugin-serve

To get started with WPS, install it first:

```
npm add webpack-plugin-serve -D
```

To integrate WPS to the project, define an npm script for launching it:

package.json

```
{
  "scripts": {
    "start": "wp --mode development",
  }
}
```

⁵<https://www.npmjs.com/package/webpack-plugin-serve>

⁶<https://www.npmjs.com/package/webpack-plugin-ramdisk>

In addition, WPS has to be connected to webpack configuration. In this case we'll run it in `liveReload` mode and refresh the browser on changes. We'll make it possible to change the port by passing an environmental variable, like `PORT=3000 npm start`:

webpack.config.js

```
const { mode } = require("webpack-nano/argv");
const {
  MiniHtmlWebpackPlugin,
} = require("mini-html-webpack-plugin");
const { WebpackPluginServe } = require("webpack-plugin-serve");

module.exports = {
  watch: mode === "development",
  entry: [".src", "webpack-plugin-serve/client"],
  mode,
  plugins: [
    new MiniHtmlWebpackPlugin({ context: { title: "Demo" } }),
    new WebpackPluginServe({
      port: parseInt(process.env.PORT, 10) || 8080,
      static: "./dist",
      liveReload: true,
      waitForBuild: true,
    }),
  ],
};
```



If you use Safari, you may have to set host: "127.0.0.1", for WebpackPluginServe for live reloading to work.

If you execute either *npm run start* or *npm start* now, you should see something similar to this in the terminal:

```
> wp --mode development
```

```
□ wps: Server Listening on: http://[::]:8080
```

```
□ webpack: asset main.js 73.1 KiB [emitted] (name: main)
  asset index.html 198 bytes [compared for emit]
  runtime modules 25.2 KiB 11 modules
  cacheable modules 25 KiB
    modules by path ./node_modules/webpack-plugin-serve/lib/client/ 23.7 KiB
  ...
    ./node_modules/webpack-plugin-serve/client.js 1.05 KiB [built] [code generated]
  0 (webpack 5.5.0) compiled successfully in 157 ms
```

The server is running, and if you open `http://localhost:8080/` at your browser, you should see a hello:

Hello world

Hello world

If you try modifying the code, you should see the output in your terminal. The browser should also perform a hard refresh so that you can see the change.



Enable the `historyFallback` flag if you are using HTML5 History API based routing.

2.4 Accessing development server from the network

To access your development server from the network, you need to figure out the IP address of your machine. For example, using `ifconfig | grep inet` on Unix, or `ipconfig` on Windows. Then you need to set your `HOST` to match your IP like this: `HOST=<ip goes here> npm start`.

2.5 Polling instead of watching files

Webpack's file watching may not work on certain systems, for example on older versions of Windows and Ubuntu.

Polling is almost mandatory when using Vagrant, Docker, or any other solution that doesn't forward events for changes on a file located in a folder shared with the virtualized machine where webpack is running. [vagrant-notify-forwarder](https://github.com/mhallin/vagrant-notify-forwarder)⁷ solves the problem for macOS and Unix.

For any of these cases, polling is a good option:

webpack.config.js

```
module.exports = {
  watchOptions: {
    aggregateTimeout: 300, // Delay the first rebuild (in ms)
    poll: 1000, // Poll using interval (in ms or a boolean)
    ignored: /node_modules/, // Ignore to decrease CPU usage
  },
};
```

The setup is more resource-intensive than the file watching, but it's worth trying out if the file watching doesn't work for you.

⁷<https://github.com/mhallin/vagrant-notify-forwarder>

2.6 Making it faster to develop webpack configuration

WPS will handle restarting the server when you change a bundled file. It's oblivious to changes made to webpack configuration, though, and you have to restart the WPS whenever you change something. The process can be automated as [discussed on GitHub](#)⁸ by using [nodemon](#)⁹ monitoring tool.

To get it to work, you have to install it first through `npm add nodemon -D`, and then set up a script:

package.json

```
{
  "scripts": {
    "watch": "nodemon --watch \"./webpack.*.*\" --exec \"npm start\"",
    "start": "wp --mode development"
  }
}
```

Development plugins

The webpack ecosystem contains many development plugins:

- [case-sensitive-paths-webpack-plugin](#)¹⁰ can be handy when you are developing on mixed environments. For example, Windows, Linux, and macOS have different expectations when it comes to path naming.
- [react-dev-utils](#)¹¹ contains webpack utilities developed for Create React App.
- [webpack-notifier](#)¹² uses system notifications to let you know of webpack status.

⁸<https://github.com/webpack/webpack-dev-server/issues/440#issuecomment-205757892>

⁹<https://www.npmjs.com/package/nodemon>

¹⁰<https://www.npmjs.com/package/case-sensitive-paths-webpack-plugin>

¹¹<https://www.npmjs.com/package/react-dev-utils>

¹²<https://www.npmjs.com/package/webpack-notifier>

2.7 Watching files outside of webpack's module graph

By default webpack only watches files that your project depends on directly, for example, when you are using **mini-html-webpack-plugin** and have customized it to load the template from a file. [webpack-add-dependency-plugin](https://www.npmjs.com/package/webpack-add-dependency-plugin)¹³ solves the problem.

2.8 Conclusion

WPS and WDS complement webpack and make it more developer-friendly. To recap:

- Webpack's watch mode is the first step towards a better development experience. You can have webpack compile bundles as you edit your source.
- WPS and WDS refresh the browser on change. They also implement *Hot Module Replacement*.
- The default webpack watching setup can be problematic on specific systems, where more resource-intensive polling is an alternative.
- WDS can be integrated into an existing Node server using a middleware, giving you more control than relying on the command line interface.
- WPS and WDS do far more than refreshing and HMR. For example, proxying allows you to connect it to other servers.

In the next chapter, you'll learn to compose configuration so that it can be developed further later in the book.

¹³<https://www.npmjs.com/package/webpack-add-dependency-plugin>

3. Composing Configuration

Even though not a lot has been done with webpack yet, the amount of configuration is starting to feel substantial. Now you have to be careful about the way you compose it as you have separate production and development targets in the project. The situation can only get worse as you want to add more functionality for testing and other purposes.

Using a single monolithic configuration file impacts comprehension and removes any potential for reusability. As the needs of your project grow, you have to figure out the means to manage webpack configuration more effectively.

3.1 Possible ways to manage configuration

You can manage webpack configuration in the following ways:

- Maintain configuration within multiple files for each environment and point webpack to each through the `--config` parameter, sharing configuration through module imports.
- Push configuration to a library, which you then consume. Examples: [webpack-config-plugins](#)¹, [Neutrino](#)², [webpack-blocks](#)³.
- Push configuration to a tool. Examples: [create-react-app](#)⁴, [kyt](#)⁵, [nwb](#)⁶.
- Maintain all configuration within a single file and branch there and rely on the `--mode` parameter. The approach is explained in detail later in this chapter.

My preferred approach is to compose webpack configuration out of smaller functions that I put together. The development of this book motivated the direction as it gives you something you can approach piece-wise while giving you a small API over webpack configuration and related techniques.

¹<https://github.com/namics/webpack-config-plugins>

²<https://neutrino.js.org/>

³<https://www.npmjs.com/package/webpack-blocks>

⁴<https://www.npmjs.com/package/create-react-app>

⁵<https://www.npmjs.com/package/kyt>

⁶<https://www.npmjs.com/package/nwb>

3.2 Composing configuration by merging

In composition based approach, you split webpack configuration and then merge it together. The problem is that a normal way of merging objects using a feature such as `Object.assign` doesn't do the right thing with arrays as if two objects have arrays attached to them, it's going to lose data. It's for this reason that I developed [webpack-merge](#)⁷.

At its core, **webpack-merge** does two things: it concatenates arrays and merges objects instead of overriding them allowing composition. The example below shows the behavior in detail:

```
> { merge } = require("webpack-merge")
...
> merge(
... { a: [1], b: 5, c: 20 },
... { a: [2], b: 10, d: 421 }
... )
{ a: [ 1, 2 ], b: 10, c: 20, d: 421 }
```

webpack-merge provides even more control through strategies that enable you to control its behavior per field. They allow you to force it to append, prepend, or replace content.

Even though **webpack-merge** was designed for this book, it has proven to be an invaluable tool beyond it. You can consider it as a learning tool and pick it up in your work if you find it handy.



webpack-chain⁸ provides a fluent API for configuring webpack allowing you to avoid configuration shape-related problems while enabling composition.

⁷<https://www.npmjs.org/package/webpack-merge>

⁸<https://www.npmjs.com/package/webpack-chain>

3.3 Setting up webpack-merge

To get started, add **webpack-merge** to the project:

```
npm add webpack-merge -D
```

To give a degree of abstraction, you can define `webpack.config.js` for higher level configuration and `webpack.parts.js` for configuration parts to consume. Here is the development server as a function:

webpack.parts.js

```
const { WebpackPluginServe } = require("webpack-plugin-serve");
const {
  MiniHtmlWebpackPlugin,
} = require("mini-html-webpack-plugin");

exports.devServer = () => ({
  watch: true,
  plugins: [
    new WebpackPluginServe({
      port: parseInt(process.env.PORT, 10) || 8080,
      static: "./dist", // Expose if output.path changes
      liveReload: true,
      waitForBuild: true,
    }),
  ],
});

exports.page = ({ title }) => ({
  plugins: [new MiniHtmlWebpackPlugin({ context: { title } })],
});
```



For the sake of simplicity, we'll develop all of the configuration using JavaScript. It would be possible to use TypeScript here as well. If you want to go that route, see the *Loading JavaScript* chapter for the required TypeScript setup.

To connect this configuration part, set up `webpack.config.js` as in the code example below:

webpack.config.js

```
const { mode } = require("webpack-nano/argv");
const { merge } = require("webpack-merge");
const parts = require("./webpack.parts");

const commonConfig = merge([
  { entry: ["../src"] },
  parts.page({ title: "Demo" }),
]);

const productionConfig = merge([]);

const developmentConfig = merge([
  { entry: ["webpack-plugin-serve/client"] },
  parts.devServer(),
]);

const getConfig = (mode) => {
  switch (mode) {
    case "production":
      return merge(commonConfig, productionConfig, { mode });
    case "development":
      return merge(commonConfig, developmentConfig, { mode });
    default:
      throw new Error(`Trying to use an unknown mode, ${mode}`);
  }
};

module.exports = getConfig(mode);
```

After these changes, the build should behave the same way as before. This time, however, you have room to expand, and you don't have to worry about how to combine different parts of the configuration.

You can add more targets by expanding the `package.json` definition and branching at `webpack.config.js` based on the need. `webpack.parts.js` grows to contain specific techniques you can then use to compose the configuration.



Webpack does not set global `NODE_ENV`⁹ based on mode by default. If you have any external tooling, such as Babel, relying on it, make sure to set it explicitly. To do this, set `process.env.NODE_ENV = mode;` within `getConfig`.

3.4 Benefits of composing configuration

There are several benefits to composing configuration:

- Splitting configuration into smaller functions lets you keep on expanding the setup.
- You can type the functions assuming you are using a language such as TypeScript.
- If you consume the configuration across multiple projects, you can publish the configuration as a package and then have only one place to optimize and upgrade as the underlying configuration changes. [SurviveJS - Maintenance](#)¹⁰ covers practices related to the approach.
- Treating configuration as a package allows you to version it as any other and deliver change logs to document the changes to the consumers.
- Taken far enough, you can end up with your own **create-react-app** that can be used to bootstrap projects quickly with your preferred setup.

3.5 Configuration layouts

In the book project, you will push all of the configuration into two files: `webpack.config.js` and `webpack.parts.js`. The former contains higher level configuration while the lower level isolates you from webpack specifics. The chosen approach allows more file layouts than the one we have.

⁹<https://github.com/webpack/webpack/issues/7074>

¹⁰<https://survivejs.com/maintenance/>

Split per configuration target

If you split the configuration per target, you could end up with a file structure as below:

```
.
└─ config
    ├── webpack.common.js
    ├── webpack.development.js
    ├── webpack.parts.js
    └─ webpack.production.js
```

In this case, you would point to the targets through `webpack --config` parameter and merge common configuration through `module.exports = merge(common, config);`.

Split parts per purpose

To add hierarchy to the way configuration parts are managed, you could decompose `webpack.parts.js` per category:

```
.
└─ config
    ├── parts
    |   ├── devserver.js
    |   ...
    |   ├── index.js
    |   └─ javascript.js
    └─ ...
```

This arrangement can make it faster to find configuration related to a category. Additionally, it can also reduce your build time if you're consuming parts from a published package as then only the required plugins will have to be loaded. A good alternative for better readability would be to arrange the functions within a single file and use comments to split it up.

Guidelines for building your own configuration packages

If you go with the configuration package approach I mentioned, consider the guidelines below:

- It can make sense to develop the package using TypeScript to document the interface well. It's particularly useful if you are authoring your configuration in TypeScript as discussed in the *Loading JavaScript* chapter.
- Expose functions that cover only one piece of functionality at a time as it lets you to replace a *Hot Module Replacement* implementation easily for example.
- Provide enough customization options through function parameters. It can be a good idea to expose an object as that lets you mimic named parameters in JavaScript. You can then destructure the parameters from that while combining this with good defaults and TypeScript types.
- Include all related dependencies within the configuration package. In specific cases you could use `peerDependencies` if you want that the consumer is able to control specific versions. Doing this means you'll likely download more dependencies that you would need but it's a good compromise.
- For parameters that have a loader string within them, use `require.resolve` to resolve against a loader within the configuration package. Otherwise the build can fail as it's looking into the wrong place for the loaders.
- When wrapping loaders, use the associated TypeScript type in parameters.
- Consider testing the package by using snapshots (`expect().toMatchSnapshot()` in Jest) to assert output changes. See the *Extending with Plugins* chapters for an example of a test harness.

3.6 Conclusion

Even though the configuration is technically the same as before, now you have room to grow it through composition.

To recap:

- Given webpack configuration is JavaScript code underneath, there are many ways to manage it.
- You should choose a method to compose configuration that makes the most sense to you. [webpack-merge](https://www.npmjs.com/package/webpack-merge)¹¹ was developed to provide a light approach for composition, but you can find many other options in the wild.
- Composition can enable configuration sharing. Instead of having to maintain a custom configuration per repository, you can share it across repositories this way. Using npm packages allows this. Developing configuration is close to developing any other code. This time, however, you codify your practices as packages.

The next parts of this book cover different techniques, and `webpack.parts.js` sees a lot of action as a result. The changes to `webpack.config.js`, fortunately, remain minimal.

¹¹<https://www.npmjs.com/package/webpack-merge>

II Styling

In this part, you will learn about styling-related concerns in detail including loading styles, separating CSS, eliminating unused CSS, and autoprefixing.

4. Loading Styles

Webpack doesn't handle styling out of the box, and you will have to use loaders and plugins to allow loading style files. In this chapter, you will set up CSS with the project and see how it works out with automatic browser refreshing. When you make a change to the CSS webpack doesn't have to force a full refresh. Instead, it can patch the CSS without one.

4.1 Loading CSS

To load CSS, you need to use [css-loader](https://www.npmjs.com/package/css-loader)¹ and [style-loader](https://www.npmjs.com/package/style-loader)².

css-loader goes through possible `@import` and `url()` lookups within the matched files and treats them as a regular ES2015 `import`. If an `@import` points to an external resource, **css-loader** skips it as only internal resources get processed further by webpack.

style-loader injects the styling through a `style` element. The way it does this can be customized. It also implements the *Hot Module Replacement* interface providing for a pleasant development experience.

The matched files can be processed through asset modules by using the `type` field at a loader definition. The feature is discussed in the *Loading Assets* part of the book.

Since inlining CSS isn't a good idea for production usage, it makes sense to use `MiniCssExtractPlugin` to generate a separate CSS file. You will do this in the next chapter.

¹<https://www.npmjs.com/package/css-loader>

²<https://www.npmjs.com/package/style-loader>

To get started, install the dependencies:

```
npm add css-loader style-loader -D
```

Add a new function at the end of the part definition:

webpack.parts.js

```
exports.loadCSS = () => ({
  module: {
    rules: [
      { test: /\.css$/, use: ["style-loader", "css-loader"] },
    ],
  },
});
```

Above means that files ending with `.css` should invoke the given loaders. Loaders return the new source files with transformations applied on them. They can be chained together like a pipe in Unix, and are evaluated from right to left:

```
styleLoader(cssLoader(input))
```

You also need to connect the fragment to the primary configuration:

webpack.config.js

```
const commonConfig = merge([
  ...
  parts.loadCSS(),
]);
```

4.2 Setting up initial CSS

You are missing the CSS still:

src/main.css

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

To make webpack aware of the CSS, we have to refer to it from our source code:

src/index.js

```
import "./main.css";  
...
```

Execute `npm start` and browse to `http://localhost:8080` if you are using the default port and open up `main.css` and change the background color to something like `lime` (`background: lime`).

Hello world

Hello cornsilk world

4.3 PostCSS

[PostCSS](http://postcss.org/)³ allows you to perform transformations over CSS through JavaScript plugins. PostCSS is the equivalent of Babel for styling and you can find plugins for many purposes. It can even fix browser bugs like `100vh` behavior on Safari [postcss-100vh-fix](https://www.npmjs.com/package/postcss-100vh-fix)⁴. PostCSS is discussed in the next chapters.

³<http://postcss.org/>

⁴<https://www.npmjs.com/package/postcss-100vh-fix>

4.4 Using CSS preprocessors

Webpack provides support for the most popular styling approaches as listed below:

- To use Less preprocessor, see [less-loader](#)⁵.
- Sass requires [sass-loader](#)⁶ or [fast-sass-loader](#)⁷ (more performant). In both cases you would add the loader after **css-loader** within the loader definition.
- For Stylus, see [stylus-loader](#)⁸.

For anything `css-in-js` related, please refer to the documentation of the specific solution. Often webpack is well supported by the options.



The *CSS Modules* appendix discusses an approach that allows you to treat local to files by default. It avoids the scoping problem of CSS.

4.5 Understanding `css-loader` lookups

To get most out of **css-loader**, you should understand how it performs its lookups. Even though the loader handles absolute and relative imports by default, it doesn't work with root relative imports - `url("/static/img/demo.png")`

If you rely on root relative imports, you have to copy the files to your project as discussed in the *Tidying Up* chapter. [copy-webpack-plugin](#)⁹ works for this purpose, but you can also copy the files outside of webpack. The benefit of the former approach is that a *Development Server* can pick that up.

Any other lookup will go through webpack and it will try to evaluate the `url` and `@import` expressions. To disable this default behavior, set **css-loader** `url: false` and `import: false` through the loader options.



[resolve-url-loader](#)¹⁰ comes in handy if you use Sass or Less. It adds support for relative imports to the environments.

⁵<https://www.npmjs.com/package/less-loader>

⁶<https://www.npmjs.com/package/sass-loader>

⁷<https://www.npmjs.com/package/fast-sass-loader>

⁸<https://www.npmjs.com/package/stylus-loader>

⁹<https://www.npmjs.com/package/copy-webpack-plugin>

¹⁰<https://www.npmjs.com/package/resolve-url-loader>

Processing css-loader imports

If you want to process **css-loader** imports in a specific way, you should set up `importLoaders` option to a number that tells the loader how many loaders before the **css-loader** should be executed against the imports found. If you import other CSS files from your CSS through the `@import` statement and want to process the imports through specific loaders, this technique is essential.

Consider the following import from a CSS file: `@import "../variables.sass";`. To process the Sass file, you would have to write configuration:

```
const config = {
  test: /\.css$/,
  use: [
    "style-loader",
    {
      loader: "css-loader",
      options: { importLoaders: 1 },
    },
    "sass-loader",
  ],
};
```

If you added more loaders, such as **postcss-loader**, to the chain, you would have to adjust the `importLoaders` option accordingly.

Loading from node_modules directory

You can load files directly from your `node_modules` directory. Consider Bootstrap and its usage for example: `@import "~bootstrap/less/bootstrap";`. The tilde character (`~`) tells webpack that it's not a relative import as by default. If tilde is included, it performs a lookup against `node_modules` (default setting) although this is configurable through the `resolve.modules`¹¹ field.

¹¹<https://webpack.js.org/configuration/resolve/#resolve-modules>



If you are using **postcss-loader**, you can skip using `~` as discussed in [postcss-loader issue tracker¹²](https://github.com/postcss/postcss-loader/issues/166). **postcss-loader** can resolve the imports without a tilde.

4.6 Conclusion

Webpack can load a variety of style formats. The approaches covered here write the styling to JavaScript bundles by default.

To recap:

- **css-loader** evaluates the `@import` and `url()` definitions. **style-loader** converts it to JavaScript and implements webpack's *Hot Module Replacement* interface.
- Webpack supports a large variety of formats compiling to CSS through loaders. These include Sass, Less, and Stylus.
- PostCSS allows you to inject functionality to CSS in through its plugin system.
- **css-loader** doesn't touch absolute nor root relative imports by default. It allows customization of loading behavior through the `importLoaders` option. You can lookup against `node_modules` by prefixing your imports with a tilde (`~`).
- Using Bootstrap with webpack requires special care. You can either go through generic loaders or a Bootstrap specific loader for more customization options.

Although the loading approach covered here is enough for development purposes, it's not ideal for production. You'll learn why and how to solve this in the next chapter by separating CSS from the source.

¹²<https://github.com/postcss/postcss-loader/issues/166>

5. Separating CSS

Even though there is a nice build set up now, where did all the CSS go? As per configuration, it has been inlined to JavaScript! Although this can be convenient during development, it doesn't sound ideal.

The current solution doesn't allow caching CSS. You can also get a **Flash of Unstyled Content** (FOUC). FOUC happens because the browser takes a while to load JavaScript, and the styles would be applied only then. Separating CSS to a file of its own avoids the problem by letting the browser to manage it separately.

Webpack provides a means to generate a separate CSS bundles using [mini-css-extract-plugin](https://www.npmjs.com/package/mini-css-extract-plugin)¹ (MCEP). It can aggregate multiple CSS files into one. For this reason, it comes with a loader that handles the extraction process. The plugin then picks up the result aggregated by the loader and emits a separate file with the styling.



It can be potentially dangerous to load inline styles with JavaScript in production as it represents an attack vector. **Critical path rendering** embraces the idea of rendering the critical CSS with inline styles in the initial HTML payload, improving the perceived performance of the site. In limited contexts inlining a small amount of CSS can be a viable option to speed up the initial load due to fewer requests.



[extract-css-chunks-webpack-plugin](https://www.npmjs.com/package/extract-css-chunks-webpack-plugin)² is a community maintained alternative to **mini-css-extract-plugin** designed especially server-side rendering in mind.

¹<https://www.npmjs.com/package/mini-css-extract-plugin>

²<https://www.npmjs.com/package/extract-css-chunks-webpack-plugin>

5.1 Setting up MiniCssExtractPlugin

Install the plugin first:

```
npm add mini-css-extract-plugin -D
```

MiniCssExtractPlugin includes a loader, `MiniCssExtractPlugin.loader` that marks the assets to be extracted. Then a plugin performs its work based on this annotation.

Add configuration as below:

webpack.parts.js

```
const MiniCssExtractPlugin = require("mini-css-extract-plugin");

exports.extractCSS = ({ options = {}, loaders = [] } = {}) => {
  return {
    module: {
      rules: [
        {
          test: /\.css$/,
          use: [
            { loader: MiniCssExtractPlugin.loader, options },
            "css-loader",
          ].concat(loaders),
          sideEffects: true,
        },
      ],
    },
    plugins: [
      new MiniCssExtractPlugin({
        filename: "[name].css",
      }),
    ],
  };
};
```


That `[name]` placeholder uses the name of the entry where the CSS is referred. Placeholders and hashing are discussed in detail in the *Adding Hashes to Filenames* chapter.



`sideEffects: true` is needed if you distribute your code as a package and want to use *Tree Shaking* against it. In most use cases, you don't have to worry about setting the flag.



If you wanted to output the resulting file to a specific directory, you could do it by passing a path. Example: `filename: "styles/[name].css"`.

Connecting with configuration

Connect the function with the configuration as below:

webpack.config.js

```
const commonConfig = merge([  
  ...  
  parts.loadCSS(),  
  parts.extractCSS(),  
]);
```



If you are using *CSS Modules*, remember to tweak use as discussed in the *Loading Styles* chapter. You can maintain separate setups for standard CSS and CSS Modules so that they get loaded through discrete logic.

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

```
> wp5-demo@0.0.0 build webpack-demo
> wp --mode production

□ webpack: Build Finished
□ webpack: asset index.html 237 bytes [compared for emit]
  asset main.js 136 bytes [compared for emit] [minimized] (name: main)
  asset main.css 33 bytes [compared for emit] (name: main)
...
webpack 5.5.0 compiled successfully in 301 ms
```

Now styling has been pushed to a separate CSS file. Thus, the JavaScript bundle has become slightly smaller, and you 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.

5.2 Managing styles outside of JavaScript

Even though referring to styling through JavaScript and then bundling is the recommended option, it's possible to achieve the same result through an entry and [globbing](#)³ the CSS files through an entry:

```
const glob = require("glob");

const commonConfig = merge([
  {
    entry: { style: glob.sync("./src/**/*.css") },
  },
]);
```

After this change, you don't have to refer to styling from your application code anymore. In this approach, you have to be careful with CSS ordering, though.

³<https://www.npmjs.com/package/glob>

If you want strict control over the ordering, you can set up a single CSS entry and then use `@import` to bring the rest to the project through it. Another option would be to set up a JavaScript entry and go through `import` to get the same effect.



[webpack-watched-glob-entries-plugin](https://www.npmjs.com/package/webpack-watched-glob-entries-plugin)⁴ provides a helper for achieving the same. As a bonus, it supports webpack's watch mode so when you modify the entries, webpack will notice.

5.3 Conclusion

The current setup separates styling from JavaScript neatly. Even though the technique is most valuable with CSS, it can be used to extract any type of modules to a separate file. The hard part of `MiniCssExtractPlugin` has to do with its setup, but the complexity can be hidden behind an abstraction.

To recap:

- Using `MiniCssExtractPlugin` with styling solves the problem of Flash of Unstyled Content (FOUC). Separating CSS from JavaScript also improves caching behavior and removes a potential attack vector.
- If you don't prefer to maintain references to styling through JavaScript, an alternative is to handle them through an entry. You have to be careful with style ordering in this case, though.

In the next chapter, you'll learn to eliminate unused CSS from the project.

⁴<https://www.npmjs.com/package/webpack-watched-glob-entries-plugin>

6. Eliminating Unused CSS

Frameworks like Bootstrap or Tailwind tend to come with a lot of CSS. Often you use only a small part of it and if you aren't careful, you will bundle the unused CSS.

[PurgeCSS¹](https://www.npmjs.com/package/purgecss) is a tool that can achieve this by analyzing files. It walks through your code and figures out which CSS classes are being used as often there is enough information for it to strip unused CSS from your project. It also works with single page applications to an extent.

Given PurgeCSS works well with webpack, we'll demonstrate it in this chapter.

6.1 Setting up Tailwind

To make the demo more realistic, let's install Tailwind to the project.

```
npm add tailwindcss postcss-loader -D
```

Generate a starter configuration using `npx tailwindcss init`. After this you'll end up with a `tailwind.config.js` file at the project root.

¹<https://www.npmjs.com/package/purgecss>

To make sure the tooling can find files containing Tailwind classes, adjust it as follows:

tailwind.config.js

```
module.exports = {  
  content: ["./src/**/*..{js}"],  
  theme: {  
    extend: {},  
  },  
  plugins: [],  
};
```

To load Tailwind, we'll have to use PostCSS:

webpack.parts.js

```
exports.tailwind = () => ({  
  loader: "postcss-loader",  
  options: {  
    postcssOptions: { plugins: [require("tailwindcss")()] },  
  },  
});
```

The new configuration still needs to be connected:

webpack.config.js

```
const cssLoaders = [parts.tailwind()];  
const commonConfig = merge([  
  ...  
  parts.extractCSS(),  
  parts.extractCSS({ loaders: cssLoaders }),  
]);
```

To make the project aware of Tailwind, import it from CSS:

src/main.css

```
@tailwind base;
@tailwind components;
/* Write your utility classes here */
@tailwind utilities;

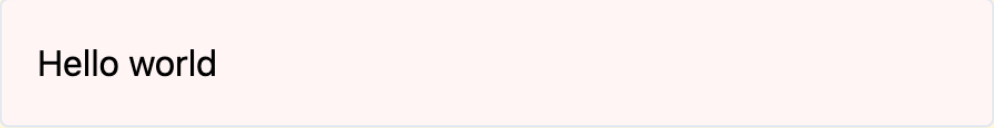
body {
  background: cornsilk;
}
```

You should also make the demo component use Tailwind classes:

src/component.js

```
export default (text = "Hello world") => {
  const element = document.createElement("div");
  element.className = "rounded bg-red-100 border max-w-md m-4 p-4";
  element.innerHTML = text;
  return element;
};
```

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



Hello world

Styled hello

Building the application (`npm run build`) should yield output:

```

[] webpack: Build Finished
[] webpack: asset main.css 1.99 MiB [emitted] [big] (name: main)
  asset index.html 237 bytes [compared for emit]
  asset main.js 193 bytes [emitted] [minimized] (name: main)
  Entrypoint main [big] 1.99 MiB = main.css 1.99 MiB main.js 193 bytes
  orphan modules 261 bytes [orphan] 2 modules
  code generated modules 309 bytes (javascript) 1.99 MiB (css/mini-extr\
act) [code generated]
    ./src/index.js + 1 modules 309 bytes [built] [code generated]
    css ./node_modules/css-loader/dist/cjs.js!./node_modules/postcss-lo\
ader/dist/cjs.js??ruleSet[1].rules[0].use[2]!./src/main.css 1.99 MiB [c\
ode generated]
```

As you can see, the size of the CSS file grew, and this is something to fix with PurgeCSS.

6.2 Enabling PurgeCSS

[purgecss-webpack-plugin](https://www.npmjs.com/package/purgecss-webpack-plugin)² allows you to eliminate most of the CSS as ideally we would bundle only the CSS classes we are using.

```
npm add glob purgecss-webpack-plugin -D
```

²<https://www.npmjs.com/package/purgecss-webpack-plugin>

You also need to configure as below:

webpack.parts.js

```
const path = require("path");
const glob = require("glob");
const PurgeCSSPlugin = require("purgecss-webpack-plugin");

const ALL_FILES = glob.sync(path.join(__dirname, "src/*.js"));

exports.eliminateUnusedCSS = () => ({
  plugins: [
    new PurgeCSSPlugin({
      paths: ALL_FILES, // Consider extracting as a parameter
      extractors: [
        {
          extractor: (content) =>
            content.match(/[\^<>"'\`\\s]*[\^<>"'\`\\s:]/g) || [],
          extensions: ["html"],
        },
      ],
    }),
  ],
});
```



For exceptions, [PurgeCSS 3.0³](https://github.com/FullHuman/purgecss/releases/tag/v3.0.0) includes **safelist** and **blocklist** options.

³<https://github.com/FullHuman/purgecss/releases/tag/v3.0.0>

Next, the part has to be connected with the configuration:

webpack.config.js

```
const productionConfig = merge();
const productionConfig = merge([parts.eliminateUnusedCSS()]);
```

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

```

[ ] webpack: Build Finished
[ ] webpack: asset main.css 7.68 KiB [emitted] (name: main)
  asset index.html 237 bytes [compared for emit]
  asset main.js 193 bytes [compared for emit] [minimized] (name: main)
  Entrypoint main 7.87 KiB = main.css 7.68 KiB main.js 193 bytes
  orphan modules 261 bytes [orphan] 2 modules
  code generated modules 309 bytes (javascript) 1.99 MiB (css/mini-extr\
act) [code generated]
    ./src/index.js + 1 modules 309 bytes [built] [code generated]
    css ./node_modules/css-loader/dist/cjs.js!./node_modules/postcss-lo\
ader/dist/cjs.js??ruleSet[1].rules[0].use[2]!./src/main.css 1.99 MiB [c\
ode generated]
  webpack 5.5.0 compiled successfully in 2429 ms
```

The size of the style has decreased noticeably. Instead of 1.99 MiB, we have roughly 7 KiB now.



Tailwind includes PurgeCSS out of the box and it can be preferable to use that. See [Tailwind documentation](https://tailwindcss.com/docs/controlling-file-size/#removing-unused-css)⁴ for more information. The example above is enough to illustrate the idea, and it works universally.



[uncss](https://www.npmjs.com/package/uncss)⁵ is a good alternative to PurgeCSS. It operates through PhantomJS and performs its work differently. You can use uncss itself as a PostCSS plugin.

⁴<https://tailwindcss.com/docs/controlling-file-size/#removing-unused-css>

⁵<https://www.npmjs.com/package/uncss>

Critical path rendering

The idea of [critical path rendering](#)⁶ takes a look at CSS performance from a different angle. Instead of optimizing for size, it optimizes for render order and emphasizes **above-the-fold** CSS. The result is achieved by rendering the page and then figuring out which rules are required to obtain the shown result.

[critical-path-css-tools](#)⁷ by Addy Osmani lists tools related to the approach.

6.3 Conclusion

Using PurgeCSS can lead to a significant decrease in file size. It's mainly valuable for static sites that rely on a massive CSS framework. The more dynamic a site or an application becomes, the harder it becomes to analyze reliably.

To recap:

- Eliminating unused CSS is possible using PurgeCSS. It performs static analysis against the source.
- The functionality can be enabled through **purgecss-webpack-plugin**.
- At best, PurgeCSS can eliminate most, if not all, unused CSS rules.
- Critical path rendering is another CSS technique that emphasizes rendering the above-the-fold CSS first. The idea is to render something as fast as possible instead of waiting for all CSS to load.

In the next chapter, you'll learn to **autoprefixer**. Enabling the feature makes it more convenient to develop complicated CSS setups that work with older browsers as well.

⁶<https://developers.google.com/web/fundamentals/performance/critical-rendering-path/>

⁷<https://github.com/addyosmani/critical-path-css-tools>

7. Autoprefixing

It can be challenging to remember which vendor prefixes you have to use for specific CSS rules to support a large variety of users. **Autoprefixing** solves this problem. It can be enabled through PostCSS and the [autoprefixer](https://www.npmjs.com/package/autoprefixer)¹ plugin. **autoprefixer** uses [Can I Use](http://caniuse.com/)² service to figure out which rules should be prefixed and its behavior can be tuned further.

7.1 Setting up autoprefixing

Achieving autoprefixing takes a small addition to the current setup. Install **postcss-loader** and **autoprefixer** first:

```
npm add postcss-loader autoprefixer -D
```

Add a fragment enabling autoprefixing:

webpack.parts.js

```
exports.autoprefix = () => ({
  loader: "postcss-loader",
  options: {
    postcssOptions: { plugins: [require("autoprefixer")()] },
  },
});
```

¹<https://www.npmjs.com/package/autoprefixer>

²<http://caniuse.com/>

To connect the loader with CSS extraction, hook it up as follows:

webpack.config.js

```
const cssLoaders = [parts.tailwind()]  
const cssLoaders = [parts.autoprefixer(), parts.tailwind()];
```



The order of the loaders matters since autoprefixing should occur after Tailwind finishes processing. The above gets evaluated as `autoprefixer(tailwind(input))`.



PostCSS supports `postcss.config.js` based configuration. It relies on [cosmiconfig](https://www.npmjs.com/package/cosmiconfig)³ internally for other formats.



[parcel-css-loader](https://www.npmjs.com/package/parcel-css-loader)⁴ is a faster alternative for PostCSS. If you don't rely on PostCSS plugins, it alone would be enough for autoprefixing your CSS.

7.2 Defining a browserslist

`autoprefixer` relies on a [browserslist](https://www.npmjs.com/package/browserslist)⁵ definition to work.

To define which browsers you want to support, set up a `.browserslistrc` file. Different tools pick up this definition, `autoprefixer` included.

³<https://www.npmjs.com/package/cosmiconfig>

⁴<https://www.npmjs.com/package/parcel-css-loader>

⁵<https://www.npmjs.com/package/browserslist>

Create a file as follows:

.browserslistrc

```
> 1% # Browser usage over 1%
Last 2 versions # Or last two versions
IE 8 # Or IE 8
```

If you build the application now (`npm run build`) and examine the built CSS, you should see that CSS was added to support older browsers. Try adjusting the definition to see what difference it makes on the build output.



You can lint CSS through [Stylelint](http://stylelint.io/)⁶. It can be set up the same way through **postcss-loader** as autoprefixing above.



It's possible to define a browserslist per development target (`BROWSERSLIST_ENV` or `NODE_ENV` in the environment) by using `[development]` kind of syntax between the declarations. See [browserslist documentation](#)⁷ for further information and options.



postcss-preset-env⁸ uses a browserslist to determine what kind of CSS to generate and which polyfills to load. You can consider it as the `@babel/preset-env` of CSS. Latter is discussed in more detail at the *Loading JavaScript* chapter.

⁶<http://stylelint.io/>

⁷<https://www.npmjs.com/package/browserslist#configuring-for-different-environments>

⁸<https://www.npmjs.com/package/postcss-preset-env>

7.3 Conclusion

Autoprefixing is a convenient technique as it decreases the amount of work needed while crafting CSS. You can maintain minimum browser requirements within a *.browserslistrc* file. The tooling can then use that information to generate optimal output.

To recap:

- Autoprefixing can be enabled through the **autoprefixer** PostCSS plugin.
- Autoprefixing writes missing CSS definitions based on your minimum browser definition.
- *.browserslistrc* is a standard file that works with tooling beyond **autoprefixer**.

III Loading Assets

In this part, you will learn how to load different types of assets using webpack's loaders. Especially images, fonts, and JavaScript receive particular attention. You also learn how webpack's loader definitions work.

8. Loader Definitions

Webpack provides multiple ways to set up module loaders. Each loader is a function accepting input and returning output. They can have side effects as they can emit to the file system and can intercept execution to implement caching.

8.1 Anatomy of a loader

Webpack supports common JavaScript formats out of the box. Other formats can be handled using **loaders** by setting up a loader, or loaders, and connecting those with your directory structure. In the example below, webpack processes JavaScript through Babel:

webpack.config.js

```
const config = {
  module: {
    rules: [
      {
        // **Conditions** to match files using RegExp, function.
        test: /\.js$/,
        // **Restrict** matching to a directory.
        include: path.join(__dirname, "app"),
        exclude: (path) => path.match(/node_modules/);
        // **Actions** to apply loaders to the matched files.
        use: "babel-loader",
      },
    ],
  },
};
```


8.2 Loader evaluation order

It's 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 as functions. You can read definition use: `["style-loader", "css-loader"]` as `style(css(input))` based on this rule. Consider the example below:

```
const config = {
  test: /\.css$/,
  use: ["style-loader", "css-loader"],
};
```

Based on the right to left rule, the example can be split up while keeping it equivalent:

```
const config = [
  { test: /\.css$/, use: "style-loader" },
  { test: /\.css$/, use: "css-loader" },
];
```



If you are not sure how a particular RegExp matches, consider using an online tool, such as [regex101](https://regex101.com/)¹, [RegExpr](http://regexpr.com/)², or [ExtendsClass RegEx Tester](https://extendsclass.com/regex-tester.html)³.

Enforcing order

Even though it would be possible to develop an arbitrary configuration using the rule above, it can be convenient to be able to force specific rules to be applied before or after regular ones. The `enforce` field can come in handy here. It can be set to either `pre` or `post` to push processing either before or after other loaders.

¹<https://regex101.com/>

²<http://regexpr.com/>

³<https://extendsclass.com/regex-tester.html>

Linting is a good example because the build should fail before it does anything else. Using `enforce: "post"` is rarer and it would imply you want to perform a check against the built source. Performing analysis against the built source is one potential example.

```
const config = {  
  test: /\.js$/,  
  enforce: "pre", // "post" too  
  use: "eslint-loader",  
};
```

It would be possible to write the same configuration without `enforce` if you chained the declaration with other loaders related to the test carefully. Using `enforce` removes the necessity for that and allows you to split loader execution into separate stages that are easier to compose.

8.3 Passing parameters to a loader

There's a query format that allows passing parameters to loaders:

```
const config = { test: /\.js$/, use: "babel-loader?presets[]=env" };
```

This style of configuration works in entries and source imports too as webpack picks it up. The format comes in handy in certain individual cases, but often you are better off using more readable alternatives. In this case, it's preferable to go through `use`:

```
const config = {  
  test: /\.js$/,  
  use: { loader: "babel-loader", options: { presets: ["env"] } },  
};
```

8.4 Inline definitions

Even though configuration level loader definitions are preferable, it's possible to write loader definitions inline:

```
import "url-loader!./foo.png"; // Process through url-loader first
import "!!url-loader!./bar.png"; // Override completely
```

The problem with this approach is that it couples your source with webpack. Nonetheless, it's still an excellent form to know.

Since configuration entries go through the same mechanism, the same forms work there as well:

```
const config = { entry: { app: "babel-loader!./app" } };
```

8.5 Branching at use using a function

In the book setup, you compose configuration on a higher level. Another option to achieve similar results would be to branch at use as webpack's loader definitions accept functions that allow you to branch depending on the environment:

```
const config = {
  test: /\.css$/,
  // `resource` refers to the resource path matched.
  // `resourceQuery` contains possible query passed to it
  // `issuer` tells about match context path
  // You have to return something falsy, object, or a string
  use: ({ resource, resourceQuery, issuer }) =>
    env === "development" && ["css-loader", "style-loader"],
};
```

Carefully applied, this technique allows different means of composition.

8.6 Loading with `info` object

Webpack provides advanced access to compilation if you pass a function as a loader definition for the `use` field. It expects you to return a loader from the call:

```
const config = {
  rules: [
    {
      test: /\.js$/,
      use: [
        (info) => ({
          loader: "babel-loader",
          options: { presets: ["env"] },
        }),
      ],
    },
  ],
};
```

If you execute code like this, you'll see a print in the console:

```
{
  resource: '/webpack-demo/src/main.css',
  realResource: '/webpack-demo/src/main.css',
  resourceQuery: '',
  issuer: '',
  compiler: 'mini-css-extract-plugin /webpack-demo/node_modules/css-loader/dist/cjs.js!/webpack-demo/node_modules/postcss-loader/src/index.js??ref--4-2!/webpack-demo/node_modules/postcss-loader/src/index.js??ref--4-3!/webpack-demo/src/main.css'
}
```

8.7 Loading based on `resourceQuery`

`oneOf` field makes it possible to route webpack to a specific loader based on a resource related match:

```
const config = {
  test: /\.png$/,
  oneOf: [
    { resourceQuery: /inline/, use: "url-loader" },
    { resourceQuery: /external/, use: "file-loader" },
  ],
};
```

If you wanted to embed the context information to the filename, the rule could use `resourcePath` over `resourceQuery`.

8.8 Loading based on `issuer`

`issuer` can be used to control behavior based on where a resource was imported. In the example below, `style-loader` is applied a CSS file is captured through JavaScript:

```
const config = {
  test: /\.css$/,
  rules: [
    { issuer: /\.js$/, use: "style-loader" },
    { use: "css-loader" },
  ],
};
```

Another approach would be to mix issuer and not:

```
const config = {
  test: /\.css$/,
  rules: [
    // Add CSS imported from other modules to the DOM
    { issuer: { not: /\.css$/ }, use: "style-loader" },
    { use: "css-loader" }, // Apply against CSS imports
  ],
};
```

8.9 Alternate ways to match files

test combined with include or exclude to constrain the match is the most common approach to match files. These accept the data types as listed below:

- test, include, exclude - Match against a RegExp, string, function, an object, or an array of conditions like these.
- resource: /inline/ - Match against a resource path including the query. Examples: /path/foo.inline.js, /path/bar.png?inline.
- issuer: /bar.js/ - Match against a resource requested from the match. Example: /path/foo.png would match if it was requested from /path/bar.js.
- resourcePath: /inline/ - Match against a resource path without its query. Example: /path/foo.inline.png.
- resourceQuery: /inline/ - Match against a resource based on its query. Example: /path/foo.png?inline.

Boolean based fields can be used to constrain these matchers further:

- not - Do **not** match against a condition (see test for accepted values).
- and - Match against an array of conditions. All must match.
- or - Match against an array while any must match.

8.10 Understanding loader behavior

Loader behavior can be understood in greater detail by inspecting them. [loader-runner](https://www.npmjs.com/package/loader-runner)⁴ allows you to run them in isolation without webpack. Webpack uses this package internally and *Extending with Loaders* chapter covers it in detail.

[inspect-loader](https://www.npmjs.com/package/inspect-loader)⁵ allows you to inspect what's being passed between loaders. Instead of having to insert `console.log`s within `node_modules`, you can attach this loader to your configuration and inspect the flow there.

8.11 Conclusion

Webpack provides multiple ways to setup loaders but sticking with `use` is enough starting from webpack 4. Be careful with loader ordering, as it's a common source of problems.

To recap:

- **Loaders** allow you determine what should happen when webpack's module resolution mechanism encounters a file.
- A loader definition consists of **conditions** based on which to match and **actions** that should be performed when a match happens.
- Webpack provides multiple ways to match and alter loader behavior. You can, for example, match based on a **resource query** after a loader has been matched and route the loader to specific actions.

In the next chapter, you'll learn to load images using webpack.

⁴<https://www.npmjs.com/package/loader-runner>

⁵<https://www.npmjs.com/package/inspect-loader>

9. Loading Images

Image loading and processing can be a concern when developing sites and applications. The problem can be solved by pushing the images to a separate service that then takes care of optimizing them and provides an interface for consuming them.

For smaller scale usage, webpack is a good option as it can both consume and process images. Doing this comes with build overhead depending on the types of operations you are performing.

Starting from webpack 5, the tool supports [asset modules](https://webpack.js.org/guides/asset-modules/)¹. Earlier dealing with assets required using loaders such as [url-loader](https://www.npmjs.com/package/url-loader)² and [file-loader](https://www.npmjs.com/package/file-loader)³ but now the functionality is integrated to webpack. The following options are supported at a loader definition:

- type: "asset/inline" emits your resources as base64 strings within the emitted assets. The process decreases the number of requests needed while growing the bundle size. The behavior corresponds with **url-loader**.
- type: "asset/resource" matches the behavior of **file-loader** and emits resources as separate files while writing references to them.
- type: "asset/source" matches [raw-loader](https://www.npmjs.com/package/raw-loader)⁴ and returns full source of the matched resource.
- type: "asset" is a mixture between asset/inline and asset/source and it will alter the behavior depending on the asset size. It's comparable to using the `limit` option of **file-loader** earlier.

`output.assetModuleFilename` field can be used to control where the assets are emitted. You could for example set it to `[hash][ext][query]` or include a directory to the path before these fragments.

¹<https://webpack.js.org/guides/asset-modules/>

²<https://www.npmjs.com/package/url-loader>

³<https://www.npmjs.com/package/file-loader>

⁴<https://www.npmjs.com/package/raw-loader>

9.1 Integrating images to the project

The syntax above can be wrapped in a small helper that can be incorporated into the book project. Set up a function as below:

webpack.parts.js

```
exports.loadImages = ({ limit } = {}) => ({
  module: {
    rules: [
      {
        test: /\..(png|jpg)$/ ,
        type: "asset",
        parser: { dataUrlCondition: { maxSize: limit } },
      },
    ],
  },
});
```

To attach it to the configuration, adjust as follows:

webpack.config.js

```
const commonConfig = merge([
  ...
  parts.loadImages({ limit: 15000 }),
]);
```

To test that the setup works, download an image or generate it (`convert -size 100x100 gradient:blue logo.png`) and refer to it from the project:

src/main.css

```
body {  
  background: cornsilk;  
  background-image: url("./logo.png");  
  background-repeat: no-repeat;  
  background-position: center;  
}
```

The behavior changes depending on the `limit` you set. Below the limit, it should inline the image while above it should emit a separate asset and a path to it.

9.2 Using `srcset`s

Modern browsers support `srcset` attribute that lets you define an image in different resolutions. The browser can then choose the one that fits the display the best. The main options are [resize-image-loader](https://www.npmjs.com/package/resize-image-loader)⁵, [html-loader-srcset](https://www.npmjs.com/package/html-loader-srcset)⁶, and [responsive-loader](https://www.npmjs.com/package/responsive-loader)⁷.

9.3 Optimizing images

In case you want to compress your images, use [image-webpack-loader](https://www.npmjs.com/package/image-webpack-loader)⁸, [svg-loader](https://www.npmjs.com/package/svg-loader)⁹ (SVG specific), or [imagemin-webpack-plugin](https://www.npmjs.com/package/imagemin-webpack-plugin)¹⁰. This type of loader should be applied first to the data, so remember to place it as the last within use listing.

Compression is particularly valuable for production builds as it decreases the amount of bandwidth required to download your image assets and speed up your site or application as a result.

⁵<https://www.npmjs.com/package/resize-image-loader>

⁶<https://www.npmjs.com/package/html-loader-srcset>

⁷<https://www.npmjs.com/package/responsive-loader>

⁸<https://www.npmjs.com/package/image-webpack-loader>

⁹<https://www.npmjs.com/package/svg-loader>

¹⁰<https://www.npmjs.com/package/imagemin-webpack-plugin>

9.4 Loading SVGs

Webpack allows a [couple ways](#)¹¹ to load SVGs. However, the easiest way is to set type as follows:

```
const config = { test: /\.svg$/, type: "asset" };
```

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

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

Consider also the following loaders:

- [svg-inline-loader](#)¹² goes a step further and eliminates unnecessary markup from your SVGs.
- [svg-sprite-loader](#)¹³ can merge separate SVG files into a single sprite, making it potentially more efficient to load as you avoid request overhead. It supports raster images (*.jpg*, *.png*) as well.
- [svg-url-loader](#)¹⁴ loads SVGs as UTF-8 encoded data urls. The result is smaller and faster to parse than base64.
- [@svgr/webpack](#)¹⁵ exposes imported SVGs as React components to consume.

9.5 Loading images dynamically

Webpack allows you to load images dynamically based on a condition. The techniques covered in the *Code Splitting* and *Dynamic Loading* chapters are enough for this purpose. Doing this can save bandwidth and load images only when you need them or preload them while you have time.

¹¹<https://github.com/webpack/webpack/issues/595>

¹²<https://www.npmjs.com/package/svg-inline-loader>

¹³<https://www.npmjs.com/package/svg-sprite-loader>

¹⁴<https://www.npmjs.com/package/svg-url-loader>

¹⁵<https://www.npmjs.com/package/@svgr/webpack>

9.6 Loading sprites

Spriting technique allows you to combine multiple smaller images into a single image. It has been used for games to describe animations and it's valuable for web development as well as you avoid request overhead.

[webpack-spritesmith](#)¹⁶ converts provided images into a sprite sheet and Sass/Less/Stylus mixins. You have to set up a SpritesmithPlugin, point it to target images, and set the name of the generated mixin. After that, your styling can pick it up:

```
@import "~sprite.sass";
```

```
.close-button {  
  sprite($close);  
}
```

```
.open-button {  
  sprite($open);  
}
```

9.7 Using placeholders

[image-trace-loader](#)¹⁷ loads images and exposes the results as image/svg+xml URL encoded data. It can be used in conjunction with **file-loader** and **url-loader** for showing a placeholder while the actual image is being loaded.

[lqip-loader](#)¹⁸ implements a similar idea. Instead of tracing, it provides a blurred image instead of a traced one.

¹⁶<https://www.npmjs.com/package/webpack-spritesmith>

¹⁷<https://www.npmjs.com/package/image-trace-loader>

¹⁸<https://www.npmjs.com/package/lqip-loader>

9.8 Referencing to images

Webpack can pick up images from style sheets through `@import` and `url()` assuming **css-loader** has been configured. You can also refer to your images within the code. In this case, you have to import the files explicitly:

```
import src from './avatar.png';

// Use the image in your code somehow now
const Profile = () => <img src={src} />;
```

Starting from webpack 5, it's possible to achieve the same as below:

```
const Profile = () => (
  <img src={new URL("./avatar.png", import.meta.url)} />
);
```

The [URL interface](https://developer.mozilla.org/en-US/docs/Web/API/URL)¹⁹ is standard and technically it would work even without a bundler assuming the image was in the correct location. If webpack is used, it will let you process the image.

It's also possible to set up dynamic imports as discussed in the *Code Splitting* chapter. Here's a small example:

```
const src = require(`./avatars/${avatar}`);
```

¹⁹<https://developer.mozilla.org/en-US/docs/Web/API/URL>

9.9 Conclusion

Webpack allows you to inline images within your bundles when needed. Figuring out proper inlining limits for your images requires experimentation. You have to balance between bundle sizes and the number of requests.

To recap:

- Use loader type field to set asset loading behavior. It replaces **file-loader** and **url-loader** used before webpack 5.
- You can find image optimization related loaders and plugins that allow you to tune their size further.
- It's possible to generate **sprite sheets** out of smaller images to combine them into a single request.
- Webpack allows you to load images dynamically based on a given condition.

You'll learn to load fonts using webpack in the next chapter.

10. Loading Fonts

Loading fonts is similar to loading images. It does come with unique challenges, though. How to know what font formats to support? There can be up to four font formats to worry about if you want to provide first class support to each browser.

The problem can be solved by deciding a set of browsers and platforms that should receive first class service. The rest can use system fonts.

You can approach the problem in several ways through webpack. You can still use the type loader field as with images. Font test patterns tend to be more complicated, though, and you have to worry about font file related lookups.

10.1 Setting up a loader

If you exclude Opera Mini, all browsers support the `.woff` format based on [Can I Use¹](https://caniuse.com/woff). `.woff2`, is widely supported by modern browsers and is another option. Going with one format, you can use a similar setup as for images and rely on `maxSize`:

```
const config = {
  test: /\.woff2?(\?v=\d+\.\d+\.\d+)?$/, // Match .woff?v=1.1.1.
  type: "asset",
  parser: { dataUrlCondition: { maxSize: 50000 } },
};
```

In case you want to make sure the site looks good on a maximum amount of browsers, you can use `type: "asset/resource"` field at a loader definition and forget about inlining. Again, it's a trade-off as you get extra requests, but perhaps it's the right move.

¹<https://caniuse.com/woff>

Here you could end up with a loader configuration as below:

```
const config = {  
  test: /\. (ttf|eot|woff|woff2)$ / ,  
  type: "asset/resource" ,  
};
```

The way you write your [CSS definition](#)² matters. To make sure you are getting the benefit from the newer formats, they should become first in the definition.

```
@font-face {  
  font-family: "Demo Font";  
  src: url("./fonts/df.woff2") format("woff2"), url("./fonts/df.woff")  
    format("woff"),  
    url("./fonts/df.eot") format("embedded-opentype"), url("./fonts/df.\  
ttf")  
    format("truetype");  
}
```

10.2 Using icon fonts

[iconfont-webpack-plugin](#)³ was designed to simplify loading icon based fonts. It inlines SVG references within CSS files.

To include only the icons that are only needed, use [fontmin-webpack](#)⁴.

10.3 Using Google Fonts

[@beyonk/google-fonts-webpack-plugin](#)⁵ can download Google Fonts to webpack build directory or connect to them using a CDN.

²<https://developer.mozilla.org/en/docs/Web/CSS/@font-face>

³<https://www.npmjs.com/package/iconfont-webpack-plugin>

⁴<https://www.npmjs.com/package/fontmin-webpack>

⁵<https://www.npmjs.com/package/@beyonk/google-fonts-webpack-plugin>

10.4 Manipulating file-loader output path and publicPath

To have more control over font output, one option is to use **url-loader** and **file-loader** as they still work. Furthermore, it's possible to manipulate `publicPath` and override the default per loader definition. The following example illustrates these techniques together:

```
{
  // Match woff2 and patterns like .woff?v=1.1.1.
  test: /\.woff2?(\.?v=\d+\.\d+\.\d+)?$/,
  use: {
    loader: "url-loader",
    options: {
      limit: 50000,
      mimetype: "application/font-woff",
      name: "./fonts/[name].[ext]", // Output below ./fonts
      publicPath: "../", // Take the directory into account
    },
  },
},
```



In the example above, the usage of **file-loader** is obscured **url-loader**. It uses **file-loader** underneath with the `limit` option. The loader options are passed to it. You can override the behavior by using the `fallback` option.

10.5 Eliminating unused characters

subfont⁶ is a tool that performs static analysis against webpack's HTML output and then rewrites the fonts to include only glyphs that are used. The subsetting process can reduce the size of the font files dramatically.

⁶<https://www.npmjs.com/package/subfont>

10.6 Generating font files based on SVGs

If you prefer to use SVG based fonts, they can be bundled as a single font file by using [webfonts-loader](https://www.npmjs.com/package/webfonts-loader)⁷.



Take care with SVG images if you have SVG specific image setup in place already. If you want to process font SVGs differently, set their definitions carefully. The *Loader Definitions* chapter covers alternatives.

10.7 Conclusion

Loading fonts is similar to loading other assets. You have to consider the browsers you want to support and choose the loading strategy based on that.

To recap:

- When loading fonts, the same techniques as for images apply. You can choose to inline small fonts while bigger ones are served as separate assets.
- If you decide to provide first class support to only modern browsers, you can select only a font format or two and let the older browsers to use system level fonts.

In the next chapter, you'll learn to load JavaScript using Babel and webpack. Webpack loads JavaScript by default, but there's more to the topic as you have to consider what browsers you want to support.

⁷<https://www.npmjs.com/package/webfonts-loader>

11. Loading JavaScript

Webpack processes ES2015 module definitions by default and transforms them into code. It does **not** transform specific syntax, such as `const`, though. The resulting code can be problematic especially in the older browsers.

To get a better idea of the default transform, we can generate a build while setting webpack's mode to none to avoid any transformation. Change the build target to use none temporarily (`{ mode: "none" }` in webpack configuration) and run `npm run build`:

dist/main.js

```
...
/***/ ((__unused_webpack_module, __webpack_exports__, __webpack_require__\n
__)) => {

  __webpack_require__.r(__webpack_exports__);
  /* harmony export */ __webpack_require__.d(__webpack_exports__, {
  /* harmony export */   "default": () => __WEBPACK_DEFAULT_EXPORT__
  /* harmony export */ });
  /* harmony default export */ const __WEBPACK_DEFAULT_EXPORT__ = ((text \n
= "Hello world") => {
    const element = document.createElement("div");
    element.className = "rounded bg-red-100 border max-w-md m-4 p-4";
    element.innerHTML = text;
    return element;
  });
  ...
}
```

The problem can be worked around by processing the code through [Babel](https://babeljs.io/)¹, a JavaScript transpiler that supports ES2015+ features and more. It resembles ESLint

¹<https://babeljs.io/>

in that it's built on top of presets and plugins. Presets are collections of plugins, and you can define your own as well.



Babel isn't the only option, although it's the most popular one. [esbuild-loader²](#) and [swc-loader³](#) are worth checking out if you don't need any specific Babel presets or plugins and want more performance.

11.1 Using Babel with webpack configuration

Even though Babel can be used standalone, as you can see in the *SurviveJS - Maintenance* book, you can hook it up with webpack as well. During development, it can make sense to skip processing if you are using language features supported by your browser.

Skipping processing is a good option, primarily if you don't rely on any custom language features and work using a modern browser. Processing through Babel becomes almost a necessity when you compile your code for production, though.

You can use Babel with webpack through [babel-loader⁴](#). It can pick up project-level Babel configuration, or you can configure it at the webpack loader itself.

Connecting Babel with a project allows you to process webpack configuration through it. Name your webpack configuration as **webpack.config.babel.js** to achieve this. [interpret⁵](#) package enables this, and it supports other tools as well.



Given that [Node supports the ES2015 specification well⁶](#) these days, you can use a lot of ES2015 features without having to process configuration through Babel.



If you use **webpack.config.babel.js**, take care with the `"modules": false`, setting. If you want to use ES2015 modules, you could skip the setting in your global Babel configuration and then configure it per environment, as discussed below.

²<https://www.npmjs.com/package/esbuild-loader>

³<https://www.npmjs.com/package/swc-loader>

⁴<https://www.npmjs.com/package/babel-loader>

⁵<https://www.npmjs.com/package/interpret>

⁶<http://node.green/>

Setting up babel-loader

The first step towards configuring Babel to work with webpack is to set up [babel-loader](#)⁷. It takes the code and turns it into a format older browsers can understand. Install **babel-loader** and include its peer dependency **@babel/core**:

```
npm add babel-loader @babel/core -D
```

As usual, let's define a function for Babel:

webpack.parts.js

```
const APP_SOURCE = path.join(__dirname, "src");

exports.loadJavaScript = () => ({
  module: {
    rules: [
      // Consider extracting include as a parameter
      { test: /\.js$/, include: APP_SOURCE, use: "babel-loader" },
    ],
  },
});
```

Next, you need to connect this to the main configuration. If you are using a modern browser for development, you can consider processing only the production code through Babel. It's used for both production and development environments in this case. Also, only application code is processed through Babel.

⁷<https://www.npmjs.com/package/babel-loader>

Adjust as below:

webpack.config.js

```
const commonConfig = merge([  
  ...  
  parts.loadJavaScript(),  
]);
```

Even though you have Babel installed and set up, you are still missing one bit: Babel configuration. The configuration can be set up using a `.babelrc` dotfile as then other tooling can use the same.

Setting up `.babelrc`

At a minimum, you need [@babel/preset-env](https://www.npmjs.com/package/@babel/preset-env)⁸. It's a Babel preset that enables the required plugins based on [browserslist](https://www.npmjs.com/package/browserslist)⁹ definition.

Install the preset first:

```
npm add @babel/preset-env -D
```

To make Babel aware of the preset, you need to write a `.babelrc`. Given webpack supports ES2015 modules out of the box, you should tell Babel to skip processing them.



See the *Autoprefixing* chapter for an expanded discussion of browserslist.

⁸<https://www.npmjs.com/package/@babel/preset-env>

⁹<https://www.npmjs.com/package/browserslist>

Here's a sample configuration:

.babelrc

```
{
  "presets": [["@babel/preset-env", { "modules": false }]]
}
```

If you execute `npm run build -- --mode none` and examine `dist/main.js`, you will see something different based on your `.browserslistrc` file. Try to include only a definition like `IE 8` there, and the code should change accordingly:

dist/main.js

```
...
/***/ ((__unused_webpack_module, __webpack_exports__, __webpack_require__\
__)) => {

  __webpack_require__.r(__webpack_exports__);
  /* harmony export */ __webpack_require__.d(__webpack_exports__, {
  /* harmony export */   "default": () => __WEBPACK_DEFAULT_EXPORT__
  /* harmony export */ });
  /* harmony default export */ const __WEBPACK_DEFAULT_EXPORT__ = (function\
on () {
    var text = arguments.length > 0 && arguments[0] !== undefined ? argum\
ents[0] : "Hello world";
    var element = document.createElement("div");
    element.className = "rounded bg-red-100 border max-w-md m-4 p-4";
    element.innerHTML = text;
    return element;
  });
  ...
```

Note especially how the function was transformed. You can try out different browser definitions and language features to see how the output changes based on the selection.



@babel/preset-env comes with a `bugfixes` option that, when enabled, writes modern syntax to one that works also in older browsers without compiling down to ES5.

11.2 Polyfilling features

@babel/preset-env allows you to polyfill certain language features for older browsers. For this to work, you should enable its `useBuiltIns` option and install [core-js¹⁰](#). If you are using async functions and want to support older browsers, then [regenerator-runtime¹¹](#) is required as well.

You have to include **core-js** to your project either through an import or an entry (`app: ["core-js", "./src"]`), except if you're using `useBuiltIns: 'usage'` to configure **@babel/preset-env**. **@babel/preset-env** rewrites the import based on your browser definition and loads only the polyfills that are needed. To learn more about **core-js** and why it's needed, [read core-js 3 release post¹²](#).



[corejs-upgrade-webpack-plugin¹³](#) makes sure you are using the newest **core-js** polyfills. Using it can help to reduce the size of the output.



core-js pollutes the global scope with objects like `Promise`. Given this can be problematic for library authors, there's [@babel/plugin-transform-runtime¹⁴](#) option. It can be enabled as a Babel plugin, and it avoids the problem of globals by rewriting the code in a such way that they aren't be needed.



Certain webpack features, such as *Code Splitting*, write `Promise` based code to webpack's bootstrap after webpack has processed loaders. The problem can be solved by applying a shim before your application code is executed. Example: entry: `{ app: ["core-js/es/promise", "./src"] }`.

¹⁰<https://www.npmjs.com/package/core-js>

¹¹<https://www.npmjs.com/package/regenerator-runtime>

¹²<https://github.com/zloirock/core-js/blob/master/docs/2019-03-19-core-js-3-babel-and-a-look-into-the-future.md>

¹³<https://www.npmjs.com/package/corejs-upgrade-webpack-plugin>

¹⁴<https://babeljs.io/docs/plugins/transform-runtime/>

11.3 Babel tips

There are other possible [.babelrc options](#)¹⁵ beyond the ones covered here. Like ESLint, `.babelrc` supports [JSON5](#)¹⁶ as its configuration format meaning you can include comments in your source, use single quoted strings, and so on.

Sometimes you want to use experimental features that fit your project. If you expect your project to live a long time, it's better to document the features you are using well.

11.4 Babel plugins

Perhaps the greatest thing about Babel is that it's possible to extend with plugins. Consider the following examples:

- [babel-plugin-import](#)¹⁷ rewrites module imports so that you can use a form such as `import { Button } from "antd";` instead of pointing to the module through an exact path.
- [babel-plugin-transform-react-remove-prop-types](#)¹⁸ removes `propTypes` related code from your production build. It also allows component authors to generate code that's wrapped so that setting environment at `DefinePlugin` can kick in as discussed in the *Environment Variables* chapter.
- [babel-plugin-macros](#)¹⁹ provides a runtime environment for small Babel modifications without requiring additional plugin setup.



It's possible to connect Babel with Node through [babel-register](#)²⁰ or [babel-cli](#)²¹. These packages can be handy if you want to execute your code through Babel without using webpack.

¹⁵<https://babeljs.io/docs/en/options>

¹⁶<https://www.npmjs.com/package/json5>

¹⁷<https://www.npmjs.com/package/babel-plugin-import>

¹⁸<https://www.npmjs.com/package/babel-plugin-transform-react-remove-prop-types>

¹⁹<https://www.npmjs.com/package/babel-plugin-macros>

²⁰<https://www.npmjs.com/package/babel-register>

²¹<https://www.npmjs.com/package/babel-cli>

11.5 Generating differential builds

To benefit from the support for modern language features and to support legacy browsers, it's possible to use webpack to generate two bundles and then write bootstrapping code that's detected by the browsers so that they use the correct ones. Doing this gives smaller bundles for modern browsers while improving JavaScript parsing time. Legacy browsers will still work as well.

As [discussed by Philip Walton²²](#), on browser-side you should use HTML like this:

```
<!-- Browsers with ES module support load this file. -->
<script type="module" src="main.mjs"></script>

<!-- Older browsers load this file (and module-supporting -->
<!-- browsers know *not* to load this file). -->
<script nomodule src="main.es5.js"></script>
```

The fallback isn't without problems as in the worst case, it can force the browser to load the module **twice**. Therefore relying on a user agent may be a better option as [highlighted by John Stewart in his example²³](#). To solve the issue, [Andrea Giammarchi has developed a universal bundle loader²⁴](#).

On webpack side, you will have to take care to generate two builds with differing browserslist definitions and names. In addition, you have to make sure the HTML template receives the script tags as above so it's able to load them.

²²<https://philipwalton.com/articles/deploying-es2015-code-in-production-today/>

²³<https://github.com/johnstew/differential-serving>

²⁴<https://medium.com/@WebReflection/a-universal-bundle-loader-6d7f3e628f93>

To give you a better idea on how to implement the technique, consider the following and set up a browserslist as below:

.browserslistrc

```
# Let's support old IE
[legacy]
IE 8

# Make this more specific if you want
[modern]
> 1% # Browser usage over 1%
```

The idea is to then write webpack configuration to control which target is chosen like this:

webpack.config.js

```
// Remember to set "mode": "production" in productionConfig
// so webpack knows to compile for the production target.
const getConfig = (mode) => {
  switch (mode) {
    case "prod:legacy":
      process.env.BROWSERSLIST_ENV = "legacy";
      return merge(commonConfig, productionConfig);
    case "prod:modern":
      process.env.BROWSERSLIST_ENV = "modern";
      return merge(commonConfig, productionConfig);
    ...
    default:
      throw new Error(`Trying to use an unknown mode, ${mode}`);
  }
};
```

Above would expect the following target:

package.json

```
{
  "scripts": {
    "build": "wp --mode prod:legacy && wp --mode prod:modern"
  }
}
```

To complete the setup, you have to write a script reference to your HTML using one of the techniques outlined above. The webpack builds can run parallel and you could use for example use the [concurrently](#)²⁵ package to speed up the execution.



These days it's possible to go one step further and [use native JavaScript modules directly in the browser](#)²⁶.

11.6 TypeScript

Microsoft's [TypeScript](#)²⁷ is a compiled language that follows a similar setup as Babel. The neat thing is that in addition to JavaScript, it can emit type definitions. A good editor can pick those up and provide enhanced editing experience. Stronger typing is valuable for development as it becomes easier to state your type contracts.

Compared to Facebook's type checker Flow, TypeScript is a safer option in terms of ecosystem. As a result, you find more premade type definitions for it, and overall, the quality of support should be better.

[ts-loader](#)²⁸ is the recommended option for TypeScript. One option is to leave only compilation to it and then handle type checking either outside of webpack or to use [fork-ts-checker-webpack-plugin](#)²⁹ to handle checking in a separate process.

²⁵<https://www.npmjs.com/package/concurrently>

²⁶<https://philipwalton.com/articles/using-native-javascript-modules-in-production-today/>

²⁷<http://www.typescriptlang.org/>

²⁸<https://www.npmjs.com/package/ts-loader>

²⁹<https://www.npmjs.com/package/fork-ts-checker-webpack-plugin>

You can also compile TypeScript with Babel through [@babel/plugin-transform-typescript](https://www.npmjs.com/package/@babel/plugin-transform-typescript)³⁰ although this comes with small [caveats](https://babeljs.io/docs/en/next/babel-plugin-transform-typescript.html#caveats)³¹.



Webpack 5 includes TypeScript support out of the box. Make sure you don't have `@types/webpack` installed in your project as it will conflict. [@types/webpack-env](https://www.npmjs.com/package/@types/webpack-env)³² contains webpack types related to the environment. If you use features like `require.context`, then you should install this one.



To split TypeScript configuration, use the `extends` property (`"extends": "./tsconfig.common"`) and then use `ts-loader` `configFile` to control which file to use through webpack.

Using TypeScript to write webpack configuration

If you have set up TypeScript to your project, you can write your configuration in TypeScript by naming the configuration file as `webpack.config.ts`. Webpack is able to detect this automatically and run it correctly.

For this to work, you need to have `ts-node`³³ or `ts-node-dev`³⁴ installed to your project as webpack uses it to execute the configuration.

If you run webpack in watch mode or through `webpack-dev-server`, by default compilation errors can cause the build to fail. To avoid this, use the following configuration:

tsconfig.json

```
{ "ts-node": { "logError": true, "transpileOnly": true } }
```

Especially the `logError` portion is important as without this `ts-node` would crash the build on error. `transpileOnly` is useful to set if you want to handle type-checking

³⁰<https://www.npmjs.com/package/@babel/plugin-transform-typescript>

³¹<https://babeljs.io/docs/en/next/babel-plugin-transform-typescript.html#caveats>

³²<https://www.npmjs.com/package/@types/webpack-env>

³³<https://www.npmjs.com/package/ts-node>

³⁴<https://www.npmjs.com/package/ts-node-dev>

outside of the process. For example, you could run `tsc` using a separate script. Often editor tooling can catch type issues as you are developing as well eliminating the need to check through `ts-node`.

11.7 WebAssembly

[WebAssembly](https://developer.mozilla.org/en-US/docs/WebAssembly)³⁵ allows developers to compile to a low-level representation of code that runs within the browser. It complements JavaScript and provides one path of potential optimization. The technology can also be useful when you want to run an old application without porting it entirely to JavaScript.

Starting from webpack 5, the tool supports new style asynchronous WebAssembly. The official examples, [wasm-simple](https://github.com/webpack/webpack/tree/master/examples/wasm-simple)³⁶ and [wasm-complex](https://github.com/webpack/webpack/tree/master/examples/wasm-complex)³⁷, illustrate the experimental functionality well. [wasm-pack's webpack tutorial](https://rustwasm.github.io/docs/wasm-pack/tutorials/hybrid-applications-with-webpack/index.html)³⁸ shows how to package Rust code using WebAssembly to be consumed through webpack.

11.8 Conclusion

Webpack loads JavaScript out of the box. Tools like Babel let you target specific browsers and have more control over the output.

To recap:

- Babel gives you control over what browsers to support. It can compile ES2015+ features to a form the older browser understand. `@babel/preset-env` is valuable as it can choose which features to compile and which polyfills to enable based on your browser definition.
- Babel allows you to use experimental language features. Babel ecosystem has numerous presets and plugins to customize it.
- Babel functionality can be enabled per development target. This way you can be sure you are using the correct plugins at the right place.

³⁵<https://developer.mozilla.org/en-US/docs/WebAssembly>

³⁶<https://github.com/webpack/webpack/tree/master/examples/wasm-simple>

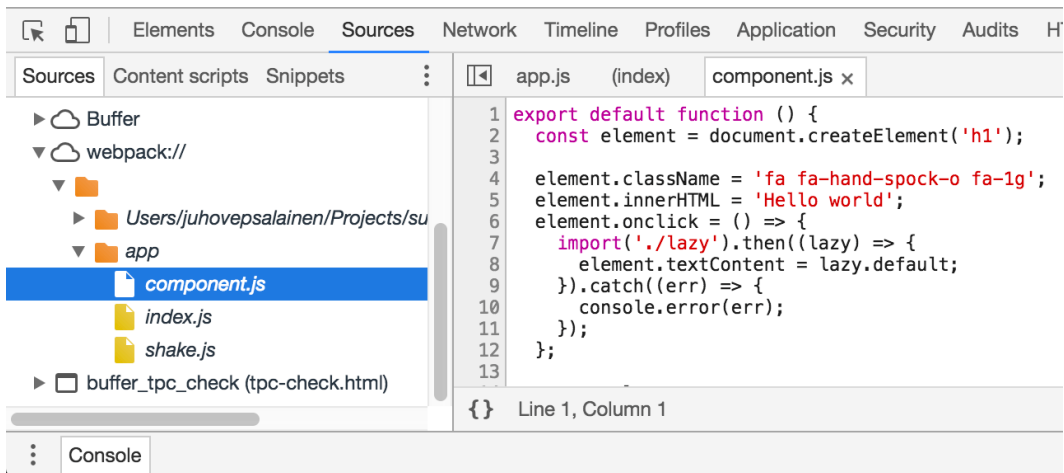
³⁷<https://github.com/webpack/webpack/tree/master/examples/wasm-complex>

³⁸<https://rustwasm.github.io/docs/wasm-pack/tutorials/hybrid-applications-with-webpack/index.html>

IV Building

In this part, you enable source maps on the build, discuss how to split it into separate bundles in various ways, and learn to tidy up the result.

12. Source Maps



Source maps in Chrome

When your source code has gone through transformations, debugging in the browser becomes a problem. **Source maps** solve this problem by providing a mapping between the original and the transformed source code. In addition to source compiling to JavaScript, this works for styling as well.

One approach is to skip source maps during development and rely on browser support of language features. If you use ES2015 without any extensions and develop using a modern browser, this can work. The advantage of doing this is that you avoid all the problems related to source maps while gaining better performance.

If you are using webpack 4 or newer and the `mode` option, the tool will generate source maps automatically for you in development mode. Production usage requires attention, though.



If you want to understand the ideas behind source maps in greater detail, see [the source map specification](https://sourcemaps.info/spec.html)¹.

¹<https://sourcemaps.info/spec.html>



To see how webpack handles source maps, see [source-map-visualization](https://sokra.github.io/source-map-visualization/)² by the author of the tool.

12.1 Inline source maps and separate source maps

Webpack can generate both inline or separate source map files. The inline ones are included to the emitted bundles and are valuable during development due to better performance. The separate files are handy for production usage as then loading source maps is optional.

It's possible you **don't** want to generate a source map for your production bundle as this makes it effortless to inspect your application. By disabling source maps, you are performing a sort of obfuscation.

Whether or not you want to enable source maps for production, they are handy for staging. Skipping source maps speeds up your build as generating source maps at the best quality can be a complicated operation.

Hidden source maps give a stack trace information only. You can connect them with a monitoring service to get traces as the application crashes allowing you to fix the problematic situations. While this isn't ideal, it's better to know about possible problems than not.

12.2 Enabling source maps

Webpack provides two ways to enable source maps. There's a `devtool` shortcut field. You can also find two plugins that give more options to tweak. The plugins are going to be discussed briefly at the end of this chapter. Beyond webpack, you also have to enable support for source maps at the browsers you are using for development.

²<https://sokra.github.io/source-map-visualization/>

Enabling source maps in webpack

To get started, you can wrap the core idea within a configuration part. You can convert this to use the plugins later if you want:

webpack.parts.js

```
exports.generateSourceMaps = ({ type }) => ({ devtool: type });
```

Webpack supports a wide variety of source map types. These vary based on quality and build speed. For now, you enable `source-map` for production and let webpack use the default for development. Set it up as follows:

webpack.config.js

```
const productionConfig = merge([  
  ...  
  parts.generateSourceMaps({ type: "source-map" }),  
]);
```

`source-map` is the slowest and highest quality option of them all, but that's fine for a production build.

If you build the project now (`npm run build`), you should see source maps in the project output at the `dist` directory. Take a good look at those `.map` files. That's where the mapping between the generated and the source happens. During development, it writes the mapping information in the bundle.

Enabling source maps in browsers

To use source maps within a browser, you have to enable source maps explicitly as per browser-specific instructions:

- [Chrome](#)³
- [Firefox](#)⁴

³<https://developers.google.com/web/tools/chrome-devtools>

⁴https://developer.mozilla.org/en-US/docs/Tools/Debugger/How_to/Use_a_source_map

- [IE Edge](#)⁵
- [Safari](#)⁶

12.3 Source map types supported by webpack

Source map types supported by webpack can be split into two categories:

- **Inline** source maps add the mapping data directly to the generated files.
- **Separate** source maps emit the mapping data to separate source map files and link the source to them using a comment. Hidden source maps omit the comment on purpose.

Thanks to their speed, inline source maps are ideal for development. Given they make the bundles big, separate source maps are the preferred solution for production. Separate source maps work during development as well if the performance overhead is acceptable.

12.4 Inline source map types

Webpack provides multiple inline source map variants. Often `eval` is the starting point and [webpack issue #2145](#)⁷ recommends `inline-module-source-map` as it's a good compromise between speed and quality while working reliably in Chrome and Firefox browsers.

To get a better idea of the available options, they are listed below while providing a small example for each. The examples are generated with the following extra webpack setup:

- `optimization.moduleIds = "named"` is set to improve readability. It's a good idea to set `optimization.chunkIds` as well in case you are using *Code Splitting*.
- `mode` is set to `false` to avoid webpack's default processing

⁵<https://docs.microsoft.com/en-us/microsoft-edge/devtools-guide/debugger#source-maps>

⁶<https://support.apple.com/guide/safari/use-the-developer-tools-in-the-develop-menu-sfri20948/mac>

⁷<https://github.com/webpack/webpack/issues/2145#issuecomment-409029231>

devtool: "eval"

eval generates code in which each module is wrapped within an eval function:

```
/***/ "./src/index.js":
/***/ (function(module, __webpack_exports__, __webpack_require__) {

  "use strict";
  eval("__webpack_require__.r(__webpack_exports__);
  \n/* harmony import */
  \nvar _main_css__WEBPACK_IMPORTED_MODULE_0__ = __webpack_require__(\"./s
  rc/main.css\");
  \n/* harmony import */
  \nvar _main_css__WEBPACK_IMPORTED_MODULE_0___default = /*#__PURE__*/__webpack_require__.n(_main_css__WEBPA
  CK_IMPORTED_MODULE_0__);
  \n/* harmony import */
  \nvar _component__WEBPACK_IMPORTED_MODULE_1__ = __webpack_require__(\"./src/component.js\");
  \n\n\\
  ndocument.body.appendChild(Object(_component__WEBPACK_IMPORTED_MODULE_1\
  __[\"default\"]));
  \n\n//# sourceMappingURL=webpack:///./src/index.js?");

  /***/ })
```

devtool: "cheap-eval-source-map"

cheap-eval-source-map goes a step further and it includes base64 encoded version of the code as a data url. The result contains only line data while losing column mappings. If you decode the resulting base64 string, you get following output:

```
{
  "version": 3,
  "file": "./src/index.js.js",
  "sources": ["webpack:///./src/index.js?3700"],
  "sourcesContent": [
    "import './main.css';
    \nimport component from './component';
    \n\\ndocument.body.appendChild(component());
  ],
  "mappings": "AAAA;AAAA;AAAA;AAAA;AAAA;AACA;AACA",
  "sourceRoot": ""
}
```

devtool: "cheap-module-eval-source-map"

cheap-module-eval-source-map is the same idea, except with higher quality and lower performance and decoding the data reveals more:

```
{
  "version": 3,
  "file": "./src/index.js.js",
  "sources": ["webpack:///./src/index.js?b635"],
  "sourcesContent": ["import './main.css';\nimport component ..."],
  "mappings": "AAAA;AAAA;AAAA;AAAA;AAAA;AACA;AAEA",
  "sourceRoot": ""
}
```

In this particular case, the difference between the options is minimal.

devtool: "eval-source-map"

eval-source-map is the highest quality option of the inline options. It's also the slowest one as it emits the most data:

```
{
  "version": 3,
  "sources": ["webpack:///./src/index.js?b635"],
  "names": ["document", "body", "appendChild", "component"],
  "mappings": "AAAA;AAAA;AAAA;AAAA;AAAA;AACA;AAEAA,QAAQ,CAACC,IAAT,CAAc\
C,WAAd,CAA0BC,0DAAS,EAAnc",
  "file": "./src/index.js.js",
  "sourcesContent": ["import './main.css';\nimport component ..."],
  "sourceRoot": ""
}
```

12.5 Separate source map types

Webpack can also generate production usage friendly source maps. These end up in separate files ending with `.map` extension and are loaded by the browser only when required. This way your users get good performance while it's easier for you to debug the application.

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

devtool: "cheap-source-map"

`cheap-source-map` is similar to the `cheap` options above. The result is going to miss column mappings. Also, source maps from loaders, such as **css-loader**, are not going to be used.

Examining the `.map` file reveals the following output in this case:

```
{
  "version": 3,
  "file": "main.js",
  "sources": [
    "webpack:///webpack/bootstrap",
    "webpack:///./src/component.js",
    "webpack:///./src/index.js",
    "webpack:///./src/main.css"
  ],
  "sourcesContent": [
    "...",
    "// extracted by mini-css-extract-plugin"
  ],
  "mappings": ";AAAA;...;;ACFA;;;A",
  "sourceRoot": ""
}
```

The source contains `//# sourceMappingURL=main.js.map` kind of comment at its end to map to this file.

devtool: "cheap-module-source-map"

cheap-module-source-map is the same as previous except source maps from loaders are simplified to a single mapping per line. It yields the following output in this case:

```
{
  "version": 3,
  "file": "main.js",
  "sources": [
    "webpack:///webpack/bootstrap",
    "webpack:///./src/component.js",
    "webpack:///./src/index.js",
    "webpack:///./src/main.css"
  ],
  "sourcesContent": [
    "...",
    "// extracted by mini-css-extract-plugin"
  ],
  "mappings": ";AAAA;...;;ACFA;;;A",
  "sourceRoot": ""
}
```



cheap-module-source-map is [currently broken if minification is used](https://github.com/webpack/webpack/issues/4176)⁸ and this is an excellent reason to avoid the option for now.

⁸<https://github.com/webpack/webpack/issues/4176>

devtool: "source-map"

source-map provides the best quality with the complete result, but it's also the slowest option. The output reflects this:

```
{
  "version": 3,
  "sources": [
    "webpack:///webpack/bootstrap",
    "webpack:///./src/component.js",
    "webpack:///./src/index.js",
    "webpack:///./src/main.css"
  ],
  "names": [
    "text",
    "element",
    "document",
    "createElement",
    "className",
    "innerHTML",
    "body",
    "appendChild",
    "component"
  ],
  "mappings": ";AAAA;...;;ACFA;;;A",
  "file": "main.js",
  "sourcesContent": [
    "...",
    "// extracted by mini-css-extract-plugin"
  ],
  "sourceRoot": ""
}
```



The [official documentation](https://webpack.js.org/configuration/devtool/#devtool)⁹ contains more information about devtool options.

⁹<https://webpack.js.org/configuration/devtool/#devtool>

devtool: "hidden-source-map"

hidden-source-map is the same as source-map except it doesn't write references to the source maps to the source files. If you don't want to expose source maps to development tools directly while you wish proper stack traces, this is handy.

devtool: "nosources-source-map"

nosources-source-map creates a source map without sourcesContent in it. You still get stack traces, though. The option is useful if you don't want to expose your source code to the client.

12.6 Other source map options

There are a couple of other options that affect source map generation:

```
const config = {
  output: {
    // Modify the name of the generated source map file.
    // You can use [file], [id], [fullhash], and [chunkhash]
    // replacements here. The default option is often enough.
    sourceMapFilename: "[file].map", // Default

    // This is the source map 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]",

    // create-react-app uses the following as it shows up well
    // in developer tools
    devtoolModuleFilenameTemplate: (info) =>
      path.resolve(info.absoluteResourcePath).replace(/\\/g, "/"),
  },
};
```

12.7 SourceMapDevToolPlugin and EvalSourceMapDevToolPlugin

If you want more control over source map generation, it's possible to use the [SourceMapDevToolPlugin](https://webpack.js.org/plugins/source-map-dev-tool-plugin/)¹⁰ or `EvalSourceMapDevToolPlugin` instead. The latter is a more limited alternative, and as stated by its name, it's handy for generating eval based source maps.

Both plugins can allow more granular control over which portions of the code you want to generate source maps for, while also having strict control over the result with `SourceMapDevToolPlugin`. Using either plugin allows you to skip the `devtool` option.

Given webpack matches only `.js` and `.css` files by default for source maps, you can use `SourceMapDevToolPlugin` to overcome this issue. This can be achieved by passing a test pattern like `/\. (js|jsx|css) ($|\?)/i`.

`EvalSourceMapDevToolPlugin` accepts only `module` field. Therefore it can be considered as an alias to `devtool: "eval"` while allowing a notch more flexibility.

12.8 Changing source map prefix

You can prefix a source map option with a **pragma** character that gets injected into the source map reference. Webpack uses `#` by default that is supported by modern browsers, so you don't have to set it.

To override this, you have to prefix your source map option with it (e.g., `@source-map`). After the change, you should see `//@` kind of reference to the source map over `//#` in your JavaScript files, assuming a separate source map type was used.

12.9 Extracting source from source maps

If a source has been minified and has source maps available, then it's possible to reconstruct the original source by using [source-from-sourcemaps](https://www.npmjs.com/package/source-from-sourcemaps)¹¹ tool. It accepts the minified source and source map as an input and then emits the source.

¹⁰<https://webpack.js.org/plugins/source-map-dev-tool-plugin/>

¹¹<https://www.npmjs.com/package/source-from-sourcemaps>

12.10 Source maps on backend

If you are using Node target with webpack as discussed in the *Build Targets* chapter, you should still generate source maps. The trick is to configure as follows:

```
const config = {
  output: {
    devtoolModuleFilenameTemplate: "[absolute-resource-path]",
  },
  plugins: [webpack.SourceMapDevToolPlugin({})],
};
```

12.11 Ignoring source map related warnings

Sometimes third-party dependencies lead to source map related warnings in the browser inspector. Webpack allows you to filter the messages as follows:

```
const config = {
  stats: {
    ignoreWarnings: { message: /Failed to parse source map/ },
  },
};
```

12.12 Using dependency source maps

Assuming you are using a package that uses inline source maps in its distribution, you can use [source-map-loader](https://www.npmjs.com/package/source-map-loader)¹² to make webpack aware of them. Without setting it up against the package, you get a minified debug output.

¹²<https://www.npmjs.com/package/source-map-loader>

12.13 Conclusion

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

To recap:

- **Source maps** can be helpful both during development and production. They provide information about what's going on and speed up debugging.
- Webpack supports many source map variants in inline and separate categories. Inline source maps are handy during development due to their speed. Separate source maps work for production as then loading them becomes optional.
- `devtool: "source-map"` is the highest quality option valuable for production.
- `inline-module-source-map` is a good starting point for development.
- Use `devtool: "hidden-source-map"` to get only stack traces during production and to send it to a third-party service for you to examine later and fix.
- `SourceMapDevToolPlugin` and `EvalSourceMapDevToolPlugin` provide more control over the result than the `devtool` shortcut.
- You should use **source-map-loader** with third-party dependencies.
- Enabling source maps for styling requires additional effort. You have to enable `sourceMap` option per styling related loader you are using.

In the next chapter, you'll learn the art of code splitting.

13. Code Splitting

Web applications tend to grow big as features are developed. The longer it takes for your site to load, the more frustrating it's to the user. This problem is amplified in a mobile environment where the connections can be slow.

Even though splitting bundles can help a notch, they are not the only solution, and you can still end up having to download a lot of data. Fortunately, it's possible to do better thanks to **code splitting** as it allows loading code lazily when you need it.

You can load more code as the user enters a new view of the application. You can also tie loading to a specific action like scrolling or clicking a button. You could also try to predict what the user is trying to do next and load code based on your guess. This way, the functionality would be already there as the user tries to access it.



Incidentally, it's possible to implement Google's [PRPL pattern](#)¹ using webpack's lazy loading. PRPL (Push, Render, Pre-cache, Lazy-load) has been designed with the mobile web in mind.



Philip Walton's [idle until urgent technique](#)² complements code splitting and lets you optimize application loading performance further. The idea is to defer work to the future until it makes sense to perform.

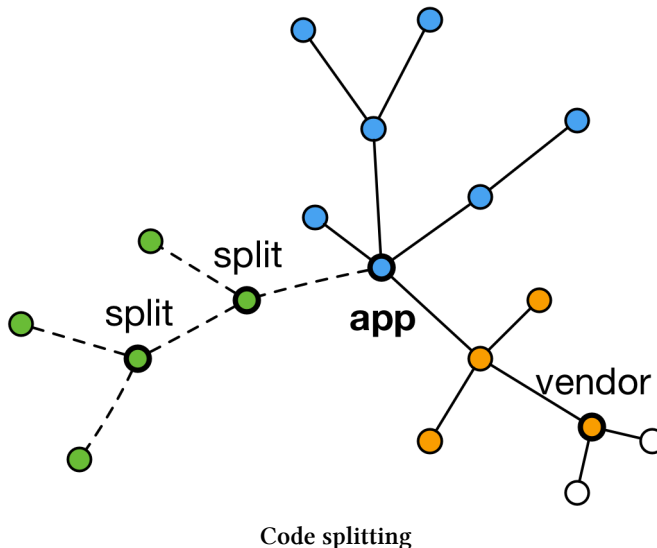
13.1 Code splitting formats

Code splitting can be done in two primary ways in webpack: through a dynamic import or `require.ensure` syntax. The latter is so called legacy syntax.

¹<https://developers.google.com/web/fundamentals/performance/prpl-pattern/>

²<https://philipwalton.com/articles/idle-until-urgent/>

The goal is to end up with a split point that gets loaded on demand. There can be splits inside splits, and you can structure an entire application based on splits. The advantage of doing this is that then the initial payload of your site can be smaller than it would be otherwise.



Dynamic import

Dynamic imports are defined as Promises:

```
import(/* webpackChunkName: "optional-name" */ "./module").then(  
  module => {...}  
)  
.catch(  
  error => {...}  
)  
;
```

Webpack provides extra control through a comment. In the example, we've renamed the resulting chunk. Giving multiple chunks the same name will group them to the same bundle. In addition `webpackMode`, `webpackPrefetch`, and `webpackPreload` are good to know options as they let you define when the import will get triggered and how the browser should treat it.

Mode lets you define what happens on `import()`. Out of the available options, `weak` is suitable for server-side rendering (SSR) as using it means the `Promise` will reject unless the module was loaded another way. In the SSR case, that would be ideal.

Prefetching tells the browser that the resource will be needed in the future while preloading means the browser will need the resource within the current page. Based on these tips the browser can then choose to load the data optimistically. [Webpack documentation explains the available options in greater detail](#)³.



[webpack.PrefetchPlugin](#)⁴ allows you to prefetch but on the level of any module.



`webpackChunkName` accepts `[index]` and `[request]` placeholders in case you want to let webpack define the name or a part of it.

The interface allows composition, and you could load multiple resources in parallel:

```
Promise.all([import("lunr"), import("../search_index.json")]).then(
  ([lunr, search]) => {
    return {
      index: lunr.Index.load(search.index),
      lines: search.lines,
    };
  }
);
```

The code above creates separate bundles to a request. If you wanted only one, you would have to use naming or define an intermediate module to import.



The syntax works only with JavaScript after configuring it the right way. If you use another environment, you may have to use alternatives covered in the following sections.

³<https://webpack.js.org/api/module-methods/#magic-comments>

⁴<https://webpack.js.org/plugins/prefetch-plugin/>

Defining a split point using a dynamic `import`

The idea can be demonstrated by setting up a module that contains a string that replaces the text of the demo button:

`src/lazy.js`

```
export default "Hello from lazy";
```

You also need to point the application to this file, so the application knows to load it by binding the loading process to click. Whenever the user happens to click the button, you trigger the loading process and replace the content:

`src/component.js`

```
export default (text = "Hello world") => {
  const element = document.createElement("div");

  element.className = "rounded bg-red-100 border max-w-md m-4 p-4";
  element.innerHTML = text;
  element.onclick = () => {
    import("./lazy")
      .then((lazy) => {
        element.textContent = lazy.default;
      })
      .catch((err) => console.error(err));

    return element;
  };
};
```

If you open up the application (`npm start`) and click the button, you should see the new text in it.

After executing `npm run build`, you should see something:

```
□ webpack: Build Finished
□ webpack: assets by status 7.95 KiB [compared for emit]
  asset main.css 7.72 KiB [compared for emit] (name: main) 1 related \
asset
  asset index.html 237 bytes [compared for emit]
  assets by status 3.06 KiB [emitted]
    asset main.js 2.88 KiB [emitted] [minimized] (name: main) 1 related \
asset
    asset 34.js 187 bytes [emitted] [minimized] 1 related asset
...
  webpack 5.5.0 compiled successfully in 3846 ms
...
```

That `34.js` is your split point. Examining the file reveals webpack has processed the code.



If you want to adjust the name of the chunk, set `output.chunkFilename`. For example, setting it to `"chunk.[id].js"` would prefix each split chunk with the word “chunk”.



If you are using TypeScript, make sure to set `compilerOptions.module` to `esnext` or `es2020` for code splitting to work correctly.

13.2 Controlling code splitting on runtime

Especially in a complex environment with third-party dependencies and an advanced deployment setup, you may want to control where split code is loaded from. [webpack-require-from](#)⁵ has been designed to address the problem, and it’s able to rewrite the import paths.

⁵<https://www.npmjs.com/package/webpack-require-from>

13.3 Code splitting in React

See [React's official documentation](#)⁶ to learn about the code splitting APIs included out of the box. The most important ones are `React.lazy` and `React.Suspense`. Currently these don't support server-side rendering. Packages like [@loadable/component](#)⁷ wrap the idea behind an interface.

13.4 Disabling code splitting

Although code splitting is a good behavior to have by default, it's not correct always, especially on server-side usage. For this reason, it can be disabled as below:

```
const config = {
  plugins: [
    new webpack.optimize.LimitChunkCountPlugin({ maxChunks: 1 }),
  ],
};
```



See [Glenn Reyes' detailed explanation](#)⁸.

13.5 Machine learning driven prefetching

Often users use an application in a specific way. The fact means that it makes sense to load specific portions of the application even before the user has accessed them. [guess-webpack](#)⁹ builds on this idea of prediction based preloading. [Minko Gechev](#) explains the approach in detail in his article¹⁰.

⁶<https://reactjs.org/docs/code-splitting.html>

⁷<https://www.npmjs.com/package/@loadable/component>

⁸<https://medium.com/@glennreyes/how-to-disable-code-splitting-in-webpack-1c0b1754a3c5>

⁹<https://www.npmjs.com/package/guess-webpack>

¹⁰<https://blog.mgechev.com/2018/03/18/machine-learning-data-driven-bundling-webpack-javascript-markov-chain-angular-react/>

13.6 Conclusion

Code splitting is a feature that allows you to push your application a notch further. You can load code when you need it to gain faster initial load times and improved user experience especially in a mobile context where bandwidth is limited.

To recap:

- **Code splitting** comes with extra effort as you have to decide what to split and where. Often, you find good split points within a router. Or you notice that specific functionality is required only when a particular feature is used. Charting is an excellent example of this.
- Use naming to pull separate split points into the same bundles.
- The techniques can be used within modern frameworks and libraries like React. You can wrap related logic to a specific component that handles the loading process in a user-friendly manner.

In the next chapter, you'll learn how to split a vendor bundle without through webpack configuration.



The *Searching with React* appendix contains a complete example of code splitting. It shows how to set up a static site index that's loaded when the user searches information.

14. Bundle Splitting

Although code splitting gives control over when code is loaded, it's not the only way webpack lets you shape the output.

Bundle splitting is a complementary technique that lets you define splitting behavior on the level of configuration. A common use case is extracting so called vendor bundle that contains third-party dependencies.

The split allows the client to download only the application bundle if there are changes only in the application code. The same goes for vendor-only changes.

To give you a quick example, instead of having `main.js` (100 kB), you could end up with `main.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.

Bundle splitting can be achieved using `optimization.splitChunks.cacheGroups`. When running in production mode, [starting from webpack 4, the tool can perform a series of splits out of the box¹](#) but in this chapter, we'll do something manually.



To invalidate the bundles correctly, you have to attach hashes to the generated bundles as discussed in the *Adding Hashes to Filenames* chapter.

14.1 Adding something to split

Given there's not much to split into the vendor bundle yet, you should add something there. Add React to the project first:

```
npm add react react-dom
```

¹<https://gist.github.com/sokra/1522d586b8e5c0f5072d7565c2bee693>

Then make the project depend on it:

src/index.js

```
import "react";
import "react-dom";
...
```

Execute `npm run build` to get a baseline build. You should end up with something as below:

```
❑ webpack: Build Finished
❑ webpack: assets by path *.js 127 KiB
  asset main.js 127 KiB [emitted] [minimized] (name: main) 2 related \
assets
  asset 34.js 187 bytes [compared for emit] [minimized] 1 related ass\
et
  asset main.css 7.72 KiB [compared for emit] (name: main) 1 related as\
set
  asset index.html 237 bytes [compared for emit]
Entrypoint main 135 KiB (323 KiB) = main.css 7.72 KiB main.js 127 KiB\
2 auxiliary assets
...
webpack 5.5.0 compiled successfully in 5401 ms
```

As you can see, `main.js` is big. That is something to fix next.

14.2 Setting up a vendor bundle

Before webpack 4, there used to be `CommonsChunkPlugin` for managing bundle splitting. The plugin has been replaced with automation and configuration.

To extract a vendor bundle from the `node_modules` directory, adjust the code as follows:

webpack.config.js

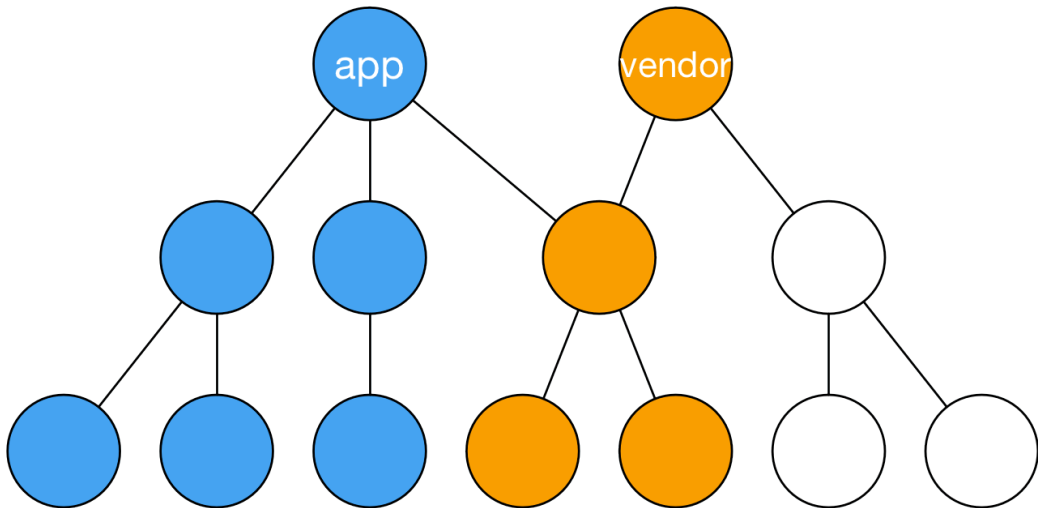
```
const productionConfig = merge([
  ...
  { optimization: { splitChunks: { chunks: "all" } } },
]);
```

If you try to generate a build now (`npm run build`), you should see something along this:

```

[ ] webpack: Build Finished
[ ] webpack: assets by status 128 KiB [emitted]
    asset 935.js 124 KiB [emitted] [minimized] (id hint: vendors) 2 related assets
    asset main.js 3.24 KiB [emitted] [minimized] (name: main) 1 related asset
    asset index.html 267 bytes [emitted]
    assets by status 7.9 KiB [compared for emit]
    asset main.css 7.72 KiB [compared for emit] (name: main) 1 related asset
    asset 34.js 187 bytes [compared for emit] [minimized] 1 related asset
    Entrypoint main 135 KiB (326 KiB) = 935.js 124 KiB main.css 7.72 KiB \
    main.js 3.24 KiB 3 auxiliary assets
    ...
    webpack 5.5.0 compiled successfully in 4847 ms
```

Now the bundles look the same as in the image below.



Main and vendor bundles after applying configuration

14.3 Controlling bundle splitting

The configuration above can be rewritten with an explicit test against `node_modules` as below to gain more control:

webpack.config.js

```
const productionConfig = merge([
  ...
  {
    optimization: {
      splitChunks: {
        cacheGroups: {
          commons: {
            test: /[\\/]node_modules[\\/]/,
            name: "vendor",
            chunks: "initial",
          },
        },
      },
    },
  },
],
```

```
    },  
  },  
},  
},  
]);
```

Starting from webpack 5, there's more control over chunking based on asset type:

```
const config = {  
  optimization: {  
    splitChunks: {  
      // css/mini-extra is injected by mini-css-extract-plugin  
      minSize: { javascript: 20000, "css/mini-extra": 10000 },  
    },  
  },  
};
```



The chunks: "initial" option doesn't apply to code-split modules while all does.

14.4 Splitting and merging chunks

Webpack provides more control over the generated chunks by two plugins:

- `AggressiveSplittingPlugin` allows you to emit more and smaller bundles. The behavior is handy with HTTP/2 due to the way the new standard works.
- `AggressiveMergingPlugin` is doing the opposite.

Here's the basic idea of aggressive splitting:

```
const config = {
  plugins: [
    new webpack.optimize.AggressiveSplittingPlugin({
      minSize: 10000,
      maxSize: 30000,
    }),
  ],
};
```

There's a trade-off as you lose out in caching if you split to multiple small bundles. You also get request overhead in HTTP/1 environment.

The aggressive merging plugin works the opposite way and allows you to combine small bundles into bigger ones:

```
const config = {
  plugins: [
    new AggressiveMergingPlugin({
      minSizeReduce: 2,
      moveToParents: true,
    }),
  ],
};
```

It's possible to get good caching behavior with these plugins if a webpack **records** are used. The idea is discussed in detail in the *Adding Hashes to Filenames* chapter.

webpack.optimize contains `LimitChunkCountPlugin` and `MinChunkSizePlugin` which give further control over chunk size.



Tobias Koppers discusses [aggressive merging in detail at the official blog of webpack²](https://medium.com/webpack/webpack-http-2-7083ec3f3ce6).

²<https://medium.com/webpack/webpack-http-2-7083ec3f3ce6>

14.5 Bundle splitting at entry configuration

Starting from webpack 5, it's possible to define bundle splitting using entries:

```
const config = {
  entry: {
    app: {
      import: path.join(__dirname, "src", "index.js"),
      dependOn: "vendor",
    },
    vendor: ["react", "react-dom"],
  },
};
```

If you have this configuration in place, you can drop `optimization.splitChunks` and the output should still be the same.



To use the approach with **webpack-plugin-serve**, you'll have to inject `webpack-plugin-serve/client` within `app.import` in this case.

14.6 Chunk types in webpack

In the example above, you used different types of webpack chunks. Webpack treats chunks in three types:

- **Entry chunks** contain webpack runtime and modules it then loads.
- **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. You generate a normal chunk in the next chapter as you set up code splitting.
- **Initial chunks** are normal chunks that count towards initial loading time of the application. As a user, you don't have to care about these. It's the split between entry chunks and normal chunks that is important.

14.7 Conclusion

The situation is better now compared to the earlier. Note how small `main` bundle compared to the `vendor` bundle. To benefit from this split, you set up caching in the next part of this book in the *Adding Hashes to Filenames* chapter.

To recap:

- Webpack allows you to split bundles from configuration entries through the `optimization.splitChunks.cacheGroups` field. It performs bundle splitting by default in production mode as well.
- A `vendor` bundle contains the third-party code of your project. The `vendor` dependencies can be detected by inspecting where the modules are imported.
- Webpack offers more control over chunking through specific plugins, such as `AggressiveSplittingPlugin` and `AggressiveMergingPlugin`. Mainly the splitting plugin can be handy in HTTP/2 oriented setups.
- Internally webpack relies on three chunk types: `entry`, `normal`, and `initial` chunks.

You'll learn to tidy up the build in the next chapter.

15. Tidying Up

The current setup doesn't clean the *build* directory between builds. As a result, it keeps on accumulating files as the project changes. Given this can get annoying, you should clean it up in between.

Another nice touch would be to include information about the build itself to the generated bundles as a small comment at the top of each file, including version information at least.

15.1 Cleaning the build directory

Starting from webpack 5.20, it supports cleaning out of the box by using the following configuration:

```
const config = {
  output: {
    clean: true,
  },
};
```

For earlier versions, you can either use [clean-webpack-plugin](https://www.npmjs.com/package/clean-webpack-plugin)¹ or solve the problem outside of webpack. You could for example trigger `rm -rf ./build && webpack` or `rimraf ./build && webpack` in an npm script to keep it cross-platform.

¹<https://www.npmjs.com/package/clean-webpack-plugin>

Setting up `output.clean`

To wrap the syntax into a function, add a function as follows.

webpack.parts.js

```
exports.clean = () => ({
  output: {
    clean: true,
  },
});
```

Connect the configuration as follows:

webpack.config.js

```
const path = require("path");

const commonConfig = merge([
  parts.clean(),
  ...
]);
```

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

15.2 Attaching a revision to the build

Attaching information related to the current build revision to the build files themselves can be used for debugging. [webpack.BannerPlugin](https://webpack.js.org/plugins/banner-plugin/)² allows you to achieve this. It can be used in combination with [git-revision-webpack-plugin](https://www.npmjs.com/package/git-revision-webpack-plugin)³ to generate a small comment at the beginning of the generated files.

²<https://webpack.js.org/plugins/banner-plugin/>

³<https://www.npmjs.com/package/git-revision-webpack-plugin>

Setting up BannerPlugin and GitRevisionPlugin

To get started, install the revision plugin:

```
npm add git-revision-webpack-plugin -D
```

Then define a part to wrap the idea:

webpack.parts.js

```
const webpack = require("webpack");
const { GitRevisionPlugin } = require("git-revision-webpack-plugin");

exports.attachRevision = () => ({
  plugins: [
    new webpack.BannerPlugin({
      banner: new GitRevisionPlugin().version(),
    }),
  ],
});
```

And connect it to the main configuration:

webpack.config.js

```
const productionConfig = merge([
  ...
  parts.attachRevision(),
]);
```

If you build the project (`npm run build`), you should notice the files ending with `.LICENSE.txt` containing comments like `/*! 0b5bb05 */` or `/*! v1.7.0-9-g5f82fe8 */` in the beginning.

The output can be customized further by adjusting the banner. You can also pass revision information to the application using `webpack.DefinePlugin`. This technique is discussed in detail in the *Environment Variables* chapter.



The code expects you run it within a Git repository! Otherwise, you get a fatal: Not a git repository (or any of the parent directories): .git error. If you are not using Git, you can replace the banner with other data.

15.3 Copying files

Copying files is another ordinary operation you can handle with webpack. [copy-webpack-plugin](#)⁴ can be handy if you need to bring external data to your build without having webpack pointing at them directly.

[cpy-cli](#)⁵ is a good option if you want to copy outside of webpack in a cross-platform way. Plugins should be cross-platform by definition.

15.4 Conclusion

Often, you work with webpack by identifying a problem and then discovering a plugin to tackle it. It's entirely acceptable to solve these types of issues outside of webpack, but webpack can often handle them as well.

To recap:

- You can find many small plugins that work as tasks and push webpack closer to a task runner.
- These tasks include cleaning the build and deployment. The *Deploying Applications* chapter discusses the latter topic in detail.
- It can be a good idea to add small comments to the production build to tell what version has been deployed. This way you can debug potential issues faster.
- Secondary tasks, like these, can be performed outside of webpack. If you are using a multi-page setup as discussed in the *Multiple Pages* chapter, this becomes a necessity.

⁴<https://www.npmjs.com/package/copy-webpack-plugin>

⁵<https://www.npmjs.com/package/cpy-cli>

V Optimizing

In this part, you will learn about code minification, setting environment variables, adding hashing to filenames, webpack runtime, analyzing build statistics, and improving webpack performance.

16. Minifying

Since webpack 4, the production output gets minified using [terser](https://www.npmjs.com/package/terser)¹ by default. Terser is an ES2015+ compatible JavaScript-minifier. Compared to UglifyJS, the earlier standard for many projects, it's a future-oriented option.

Although webpack minifies the output by default, it's good to understand how to customize the behavior should you want to adjust it further or replace the minifier.

16.1 Minifying JavaScript

The point of **minification** is to convert the code into a smaller form. Safe **transformations** do this without losing any meaning by rewriting code. Good examples of this include renaming variables or even removing entire blocks of code based on the fact that they are unreachable (`if (false)`).

Unsafe transformations can break code as they can lose something implicit the underlying code relies upon. For example, Angular 1 expects specific function parameter naming when using modules. Rewriting the parameters breaks code unless you take precautions against it in this case.

Modifying JavaScript minification process

In webpack, minification process is controlled through two configuration fields: `optimization.minimize` flag to toggle it and `optimization.minimizer` array to configure the process.

To tune the defaults, we'll attach [terser-webpack-plugin](https://www.npmjs.com/package/terser-webpack-plugin)² to the project so that it's possible to adjust it.

¹<https://www.npmjs.com/package/terser>

²<https://www.npmjs.com/package/terser-webpack-plugin>

To get started, include the plugin to the project:

```
npm add terser-webpack-plugin -D
```

To attach it to the configuration, define a part for it first:

webpack.parts.js

```
const TerserPlugin = require("terser-webpack-plugin");

exports.minifyJavaScript = () => ({
  optimization: { minimizer: [new TerserPlugin()] },
});
```

Hook it up to the configuration:

webpack.config.js

```
const productionConfig = merge([
  parts.minifyJavaScript(),
  ...
]);
```

If you execute `npm run build` now, you should see result close to the same as before.



Source maps are disabled by default. You can enable them through the `sourceMap` flag. You should check **terser-webpack-plugin** documentation for further options.



To adjust Terser behavior, attach `terserOptions` with the related options to the plugin.

16.2 Speeding up JavaScript execution

Specific solutions allow you to preprocess code so that it will run faster. They complement the minification technique and can be split into **scope hoisting**, **pre-evaluation**, and **improving parsing**. It's possible these techniques grow overall bundle size sometimes while allowing faster execution.

Scope hoisting

Since webpack 4, it applies scope hoisting in production mode by default. It hoists all modules to a single scope instead of writing a separate closure for each. Doing this slows down the build but gives you bundles that are faster to execute. [Read more about scope hoisting](#)³ at the webpack blog.



Set `stats.optimizationBailout` flag as `true` to gain debugging information related to hoisting results.

16.3 Minifying HTML

If you consume HTML templates through your code using [html-loader](#)⁴, you can preprocess it through [posthtml](#)⁵ with [posthtml-loader](#)⁶. You can use [posthtml-minifier](#)⁷ to minify your HTML through it and [posthtml-minify-classnames](#)⁸ to reduce the length of class names.

³<https://medium.com/webpack/brief-introduction-to-scope-hoisting-in-webpack-8435084c171f>

⁴<https://www.npmjs.com/package/html-loader>

⁵<https://www.npmjs.com/package/posthtml>

⁶<https://www.npmjs.com/package/posthtml-loader>

⁷<https://www.npmjs.com/package/posthtml-minifier>

⁸<https://www.npmjs.com/package/posthtml-minify-classnames>

16.4 Minifying CSS

[css-minimizer-webpack-plugin](https://www.npmjs.com/package/css-minimizer-webpack-plugin)⁹ is a plugin-based option that applies a chosen minimifier on CSS assets. Using `MiniCssExtractPlugin` can lead to duplicated CSS given it only merges text chunks. **css-minimizer-webpack-plugin** avoids this problem by operating on the generated result and thus can lead to a better outcome. The plugin uses [cssnano](http://cssnano.co/)¹⁰ underneath.

Setting Up CSS minification

To get started, install **css-minimizer-webpack-plugin** first:

```
npm add css-minimizer-webpack-plugin -D
```

Like for JavaScript, you can wrap the idea in a configuration part:

webpack.parts.js

```
const CssMinimizerPlugin = require("css-minimizer-webpack-plugin");

exports.minifyCSS = ({ options }) => ({
  optimization: {
    minimizer: [
      new CssMinimizerPlugin({ minimizerOptions: options }),
    ],
  },
});
```



To override **cssnano** with another option, use the `minify` option. It accepts a function with the signature `(data, inputMap, minimizerOptions) => <string>`.

⁹<https://www.npmjs.com/package/css-minimizer-webpack-plugin>

¹⁰<http://cssnano.co/>

Then, connect with the main configuration:

webpack.config.js

```
const productionConfig = merge([
  parts.minifyJavaScript(),
  parts.minifyCSS({ options: { preset: ["default"] } }),
  ...
]);
```

If you build the project now (`npm run build`), you should notice that CSS has become smaller as it's missing comments and has been concatenated:

```
webpack: Build Finished
webpack: assets by path *.js 129 KiB
  asset vendor.js 126 KiB [emitted] [minimized] (name: vendor) (id hi\
nt: commons) 2 related assets
  asset main.js 3.32 KiB [emitted] [minimized] (name: main) 2 related\
assets
  asset 34.js 247 bytes [emitted] [minimized] 2 related assets
  asset main.css 730 bytes [emitted] (name: main)
...
webpack 5.5.0 compiled successfully in 6388 ms
```



Using [last-call-webpack-plugin](https://www.npmjs.com/package/last-call-webpack-plugin)¹¹ is a more generic approach and you can use it to define which processing to use against which file format before webpack finishes processing.

¹¹<https://www.npmjs.com/package/last-call-webpack-plugin>

16.5 Compressing bundles

Compression techniques, such as gzip or brotli, can be used to reduce the file size further. The downside of using additional compression is that it will lead to extra computation on the client side but on the plus side you save bandwidth.

Often the compression setup can be done on server-side. Using webpack, it's possible to perform preprocessing with [compression-webpack-plugin](#)¹².

16.6 Obfuscating output

To make it more tricky for third parties to use your code, use [webpack-obfuscator](#)¹³. Although protecting code is difficult when it's shared with the client, the code can be made much harder to use.

16.7 Conclusion

Minification is the most comfortable step you can take to make your build smaller. To recap:

- **Minification** process analyzes your source code and turns it into a smaller form with the same meaning if you use safe transformations. Specific unsafe transformations allow you to reach even smaller results while potentially breaking code that relies, for example, on exact parameter naming.
- Webpack performs minification in production mode using Terser by default.
- Besides JavaScript, it's possible to minify other assets, such as CSS and HTML too. Minifying these requires specific technologies that have to be applied through loaders and plugins of their own.

You'll learn to apply tree shaking against code in the next chapter.

¹²<https://www.npmjs.com/package/compression-webpack-plugin>

¹³<https://www.npmjs.com/package/webpack-obfuscator>

17. Tree Shaking

Tree shaking is a feature enabled by the ES2015 module definition. The idea is that given it's possible to analyze the module definition statically without running it, webpack can tell which parts of the code are being used and which are not. It's possible to verify this behavior by expanding the application and adding code there that should be eliminated.

Starting from webpack 5, tree shaking has been improved and it works in cases where it didn't work before, including nesting and CommonJS.

17.1 Demonstrating tree shaking

To shake code, you have to define a module and use only a part of its code:

src/shake.js

```
const shake = () => console.log("shake");
const bake = () => console.log("bake");

export { shake, bake };
```

To make sure you use a part of the code, alter the application entry point:

src/index.js

```
...
import { bake } from "./shake";

bake();
```

If you build the project again (`npm run build`) and examine the build (`dist/main.js`), it should contain `console.log("bake")`, but miss `console.log("shake")`. That's tree shaking in action.



To understand which exports are being shaken out, set `stats.usedExports` field to `true` in webpack configuration.



For tree shaking to work with TypeScript, you have to set `compilerOptions.module` to `es2015` or equivalent. The idea is to retain ES2015 module definitions for webpack to process as it needs the information for tree shaking.

17.2 Tree shaking on package level

The same idea works with dependencies that use the ES2015 module definition. Given the related packaging, standards are still emerging, you have to be careful when consuming such packages. Webpack tries to resolve `package.json` `module` field for this reason.

For tools like webpack to allow tree shake npm packages, you should generate a build that has transpiled everything else except the ES2015 module definitions and then point to it through `package.json` `module` field. In Babel terms, you have to let webpack to manage ES2015 modules by setting `"modules": false`.

Another important point is to set `"sideEffects": false` to state that when the code is executing, it doesn't modify anything outside of its own scope. The property also accepts an array of file paths if you want to be more specific. The [Stack Overflow question related to this explains in detail why](https://stackoverflow.com/questions/49160752/what-does-webpack-4-expect-from-a-package-with-sideeffects-false)¹.

¹<https://stackoverflow.com/questions/49160752/what-does-webpack-4-expect-from-a-package-with-sideeffects-false>

17.3 Tree shaking with external packages

To get most out of tree shaking with external packages, you have to use [babel-plugin-transform-imports](https://www.npmjs.com/package/babel-plugin-transform-imports)² to rewrite imports so that they work with webpack's tree shaking logic. See [webpack issue #2867](https://github.com/webpack/webpack/issues/2867)³ for more information.

It's possible to force `"sideEffects": false` at webpack configuration by setting up a loader definition with `test: path.resolve(__dirname, "node_modules/package")` and `sideEffects: false` fields.



[SurviveJS - Maintenance](https://survivejs.com/maintenance/)⁴ delves deeper to the topic from the package point of view.

17.4 Conclusion

Tree shaking is a potentially powerful technique. For the source to benefit from tree shaking, npm packages have to be implemented using the ES2015 module syntax, and they have to expose the ES2015 version through `package.json` `module` field tools like webpack can pick up.

To recap:

- **Tree shaking** drops unused pieces of code based on static code analysis. Webpack performs this process for you as it traverses the dependency graph.
- To benefit from tree shaking, you have to use ES2015 module definition.
- As a package author, you can provide a version of your package that contains ES2015 modules, while the rest has been transpiled to ES5. It's important to set `"sideEffects": false` as after that webpack knows it's safe to tree shake the package.

You'll learn how to manage environment variables using webpack in the next chapter.

²<https://www.npmjs.com/package/babel-plugin-transform-imports>

³<https://github.com/webpack/webpack/issues/2867>

⁴<https://survivejs.com/maintenance/packaging/building/>

18. Environment Variables

Sometimes a part of your code should execute only during development. Or you could have experimental features in your build that are not ready for production yet. Controlling **environment variables** becomes valuable as you can toggle functionality using them.

Since JavaScript minifiers can remove dead code (`if (false)`), you can build on top of this idea and write code that gets transformed into this form. Webpack's `DefinePlugin` enables replacing **free variables** so that you can convert

```
if (process.env.NODE_ENV === "development") {  
  console.log("Hello during development");  
}
```

kind of code to `if (true)` or `if (false)` depending on the environment.

You can find packages that rely on this behavior. React is perhaps the most known example of an early adopter of the technique. Using `DefinePlugin` can bring down the size of your React production build somewhat as a result, and you can see a similar effect with other packages as well.

Starting from webpack 4, `process.env.NODE_ENV` is set within the build based on the given mode but not globally. To pass the variable to other tools, you'll have to set it explicitly outside of webpack or within webpack configuration.



`webpack.EnvironmentPlugin(["NODE_ENV"])` is a shortcut that allows you to refer to environment variables. It uses `DefinePlugin` underneath, and you can achieve the same effect by passing `process.env.NODE_ENV`.



[dotenv-webpack](https://www.npmjs.com/package/dotenv-webpack)¹ goes a step further and maps environment variables from a dotfile (`.env`) to a build using `DefinePlugin` underneath.

¹<https://www.npmjs.com/package/dotenv-webpack>

18.1 The basic idea of DefinePlugin

To understand the idea of DefinePlugin better, consider the example below:

```
var foo;
if (foo === "bar") console.log("bar"); // Not free
if (bar === "bar") console.log("bar"); // Free
```

If you replaced `bar` with a string like `"foobar"`, then you would end up with the code as below:

```
var foo;
if (foo === "bar") console.log("bar"); // Not free
if ("foobar" === "bar") console.log("bar");
```

Further analysis shows that `"foobar" === "bar"` equals `false` so a minifier gives the following:

```
var foo;
if (foo === "bar") console.log("bar"); // Not free
if (false) console.log("bar");
```

A minifier eliminates the `if` statement as it has become dead code:

```
var foo;
if (foo === "bar") console.log("bar"); // Not free
// if (false) means the block can be dropped entirely
```

Elimination is the core idea of DefinePlugin and it allows toggling. A minifier performs analysis and toggles entire portions of the code.



[babel-plugin-transform-define](https://www.npmjs.com/package/babel-plugin-transform-define)² achieves similar behavior with Babel.

²<https://www.npmjs.com/package/babel-plugin-transform-define>

18.2 Setting `process.env.NODE_ENV`

As before, encapsulate this idea to a function. Due to the way webpack replaces the free variable, you should push it through `JSON.stringify`. You end up with a string like `'"demo"'` and then webpack inserts that into the slots it finds:

`webpack.parts.js`

```
exports.setFreeVariable = (key, value) => {  
  const env = {};  
  env[key] = JSON.stringify(value);  
  
  return {  
    plugins: [new webpack.DefinePlugin(env)],  
  };  
};
```

Connect this with the configuration:

`webpack.config.js`

```
const commonConfig = merge([  
  ...  
  parts.setFreeVariable("HELLO", "hello from config"),  
]);
```

Finally, add something to replace:

`src/component.js`

```
export default (text = "Hello world") => {  
export default (text = HELLO) => {  
  const element = document.createElement("div");  
  ...  
};
```

If you run the application, you should see a new message on the button.

18.3 Choosing which module to use

The techniques discussed in this chapter can be used to choose entire modules depending on the environment. As seen above, `DefinePlugin` based splitting allows you to choose which branch of code to use and which to discard. This idea can be used to implement branching on module level. Consider the file structure below:

```
.
├─ store
│  ├── index.js
│  ├── store.dev.js
│  └─ store.prod.js
```

The idea is that you choose either dev or prod version of the store depending on the environment. It's that `index.js` which does the hard work:

```
if (process.env.NODE_ENV === "production") {
  module.exports = require("./store.prod");
} else {
  module.exports = require("./store.dev");
}
```

Webpack can pick the right code based on the `DefinePlugin` declaration and this code. You have to use CommonJS module definition style here as ES2015 imports don't allow dynamic behavior by design.



A related technique, **aliasing**, is discussed in the *Consuming Packages* chapter.



You have to be careful when doing a check against `process.env.NODE_ENV` in complex pieces of code. [Johnny Reilly](https://blog.johnnyreilly.com/2018/03/its-not-dead-webpack-and-dead-code.html) gives a good example of a [problematic case](#)³.

³<https://blog.johnnyreilly.com/2018/03/its-not-dead-webpack-and-dead-code.html>

18.4 Conclusion

Setting environment variables is a technique that allows you to control which paths of the source are included in the build.

To recap:

- Webpack allows you to set **environment variables** through `DefinePlugin` and `EnvironmentPlugin`. Latter maps the system level environment variables to the source.
- `DefinePlugin` operates based on **free variables** and it replaces them as webpack analyzes the source code. You can achieve similar results by using Babel plugins.
- Given minifiers eliminate dead code, using the plugins allows you to remove the code from the resulting build.
- The plugins enable module level patterns. By implementing a wrapper, you can choose which file webpack includes to the resulting build.
- In addition to these plugins, you can find other optimization related plugins that allow you to control the build result in many ways.

To ensure the build has good cache invalidation behavior, you'll learn to include hashes to the generated filenames in the next chapter. This way the client notices if assets have changed and can fetch the updated versions.

19. Adding Hashes to Filenames

Even though the generated build works, the file names it uses is problematic. It doesn't allow to leverage client level cache efficiently as there's no way tell whether or not a file has changed. Cache invalidation can be achieved by including a hash to the filenames.



Starting from version 5, webpack is using a deterministic way of generating filenames that's a good compromise between bundle size and long time caching. The behavior is controllable through `optimization.moduleIds` and `optimization.chunkIds`. Latter applies to *Code Splitting*.



As discussed by Jake Archibald¹, deploying code or bundle split source comes with inherent challenges. It's possible the user is running an old version. The problem is detecting the case and dealing with it. One option would be to ask the user to refresh the site/application for example.

19.1 Placeholders

Webpack provides **placeholders** for this purpose. These strings are used to attach specific information to webpack output. The most valuable ones are:

- `[id]` - Returns the chunk id.
- `[path]` - Returns the file path.
- `[name]` - Returns the file name.
- `[ext]` - Returns the extension. `[ext]` works for most available fields.
- `[fullhash]` - Returns the build hash. If any portion of the build changes, this changes as well.

¹<https://jakearchibald.com/2020/multiple-versions-same-time/>

- `[chunkhash]` - Returns an entry chunk-specific hash. Each entry defined in the configuration receives a hash of its own. If any portion of the entry changes, the hash will change as well. `[chunkhash]` is more granular than `[fullhash]` by definition.
- `[contenthash]` - Returns a hash generated based on content. It's the new default in production mode starting from webpack 5.

It's preferable to use particularly `hash` and `contenthash` only for production purposes as hashing doesn't do much good during development.



There are more options available, and you can even modify the hashing and digest type as discussed at [loader-utils²](#) documentation.



If you are using webpack 4, be careful with `contenthash` as [it's not fully reliable³](#). There `chunkhash` may be the preferable option.

Example placeholders

Assume you have the following configuration:

```
const config = {
  output: {
    path: PATHS.build,
    filename: "[name].[contenthash].js",
  },
};
```

²<https://www.npmjs.com/package/loader-utils>

³<https://github.com/webpack/webpack/issues/11146>

Webpack generates filenames like these based on it:

```
main.d587bbd6e38337f5accd.js  
vendor.dc746a5db4ed650296e1.js
```

If the file contents related to a chunk are different, the hash changes as well, thus the cache gets invalidated. More accurately, the browser sends a new request for the file. If only main bundle gets updated, only that file needs to be requested again.

The same result can be achieved by generating static filenames and invalidating the cache through a querystring (i.e., `main.js?d587bbd6e38337f5accd`). The part behind the question mark invalidates the cache. According to [Steve Souders⁴](http://www.stevesouders.com/blog/2008/08/23/revving-filenames-dont-use-querystring/), attaching the hash to the filename is the most performant option.

19.2 Setting up hashing

The build needs tweaking to generate proper hashes. Adjust as follows:

webpack.config.js

```
const productionConfig = merge([  
  {  
    output: {  
      chunkFilename: "[name].[contenthash].js",  
      filename: "[name].[contenthash].js",  
      assetModuleFilename: "[name].[contenthash][ext][query]",  
    },  
  },  
  ...  
]);
```

⁴<http://www.stevesouders.com/blog/2008/08/23/revving-filenames-dont-use-querystring/>

To make sure extracted CSS receives hashes as well, adjust:

webpack.parts.js

```
exports.extractCSS = ({ options = {}, loaders = [] } = {}) => {
  return {
    ...
    plugins: [
      new MiniCssExtractPlugin({
      filename: "[name].css",
        filename: "[name].[contenthash].css",
      }),
    ],
  };
};
```

If you generate a build now (`npm run build`), you should see something:

```
❏ webpack: Build Finished
❏ webpack: assets by path *.js 129 KiB
  asset vendor.16...22.js 126 KiB [emitted] [immutable] [minimized] (\
name: vendor) (id hint: commons) 2 related assets
  asset main.db...11.js 3.4 KiB [emitted] [immutable] [minimized] (na\
me: main) 2 related assets
  asset 34.a4...c5.js 257 bytes [emitted] [immutable] [minimized] 2 r\
elated assets
  asset main.bd...ca.css 1.87 KiB [emitted] [immutable] (name: main)
  asset index.html 285 bytes [emitted]
...
webpack 5.5.0 compiled successfully in 6593 ms
```

The files have neat hashes now. To prove that it works for styling, you could try altering `src/main.css` and see what happens to the hashes when you rebuild.

19.3 Conclusion

Including hashes related to the file contents to their names allows to invalidate them on the client-side. If a hash has changed, the client is forced to download the asset again.

To recap:

- Webpack’s **placeholders** allow you to shape filenames and enable you to include hashes to them.
- The most valuable placeholders are `[name]`, `[contenthash]`, and `[ext]`. A content hash is derived based on the chunk content.
- If you are using `MiniCssExtractPlugin`, you should use `[contenthash]` as well. This way the generated assets get invalidated only if their content changes.

The next chapter discusses the topic of webpack **runtime**. To make sure changes made to it won’t invalidate more code than it should, it’s a good practice to separate it.



Philip Walton has written about caching in detail⁵ and his article is a great read if you want to know more about the topic on a more general level.

⁵<https://philipwalton.com/articles/cascading-cache-invalidation/>

20. Separating a Runtime

When webpack writes bundles, it maintains a **runtime** as well. The runtime includes a manifest of the files to be loaded initially. If the names of the files change, then the manifest changes and the change invalidates the file in which it is contained. For this reason, it can be a good idea to write the runtime to a file of its own or inline the manifest information to the `index.html` file of the project.

20.1 Extracting a runtime

Most of the work was done already when `extractBundles` was set up in the *Bundle Splitting* chapter. To extract the runtime, define `optimization.runtimeChunk` as follows:

webpack.config.js

```
const productionConfig = merge([
  ...
  —{ optimization: { splitChunks: { chunks: "all" } } },
  {
    optimization: {
      splitChunks: { chunks: "all" },
      runtimeChunk: { name: "runtime" },
    },
  },
  ...
]);
```

The name `runtime` is used by convention. You can use any other name, and it will still work.

If you build the project now (`npm run build`), you should see something:

```

❑ webpack: Build Finished
❑ webpack: assets by path *.js 130 KiB
  asset vendor.16...22.js 126 KiB [emitted] [immutable] [minimized] (\
name: vendor) (id hint: commons) 2 related assets
  asset runtime.41...f8.js 3.01 KiB [emitted] [immutable] [minimized]\
(name: runtime) 2 related assets
  asset main.ed...dd.js 633 bytes [emitted] [immutable] [minimized] (\
name: main) 2 related assets
  asset 34.a4...c5.js 257 bytes [emitted] [immutable] [minimized] 2 r\
elated assets
  asset main.ac...a1.css 1.87 KiB [emitted] [immutable] (name: main)
  asset index.html 324 bytes [emitted]
...
webpack 5.5.0 compiled successfully in 7209 ms

```

This change gave a separate file that contains the runtime. In the output above it has been marked with `runtime` chunk name. As the setup is using `MiniHtmlWebpackPlugin`, there is no need to worry about loading the runtime ourselves as the plugin adds a reference to `index.html`. Try adjusting `src/index.js` and see how the hashes change.

Starting from webpack 5, the tool will take your `browserslist` definition into account when generating the runtime. See the *Autoprefixing* chapter for an expanded discussion. In webpack 5, it's possible to use `target` to define in which format the runtime is written. Setting it to `es5` would emit ECMAScript 5 compatible code while setting to `es2015` would generate shorter code for the newer target. The setting also affects the *Minifying* process.



To get a better idea of the runtime contents, run the build in development mode or pass `none` to `mode` through configuration. You should see something familiar there.

20.2 Using records

As hinted in the *Bundle Splitting* chapter, `AggressiveSplittingPlugin` and others use **records** to implement caching. The approaches discussed above are still valid, but records go one step further.

Records are used for storing module IDs across separate builds. The problem is that you need to save this file. If you build locally, one option is to include it in your version control.

To generate a `records.json` file, adjust the configuration as follows:

webpack.config.js

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

...

const productionConfig = merge([
  ...
  { recordsPath: path.join(__dirname, "records.json") },
]);
```

If you build the project (`npm run build`), you should see a new file, `records.json`, at the project root. The next time webpack builds, it picks up the information and rewrites the file if it has changed.

Records are particularly valuable if you have a complicated setup with code splitting and want to make sure the split parts gain correct caching behavior. The biggest problem is maintaining the record file.



`recordsInputPath` and `recordsOutputPath` give more granular control over input and output, but often setting only `recordsPath` is enough.



If you change the way webpack handles module IDs, possible existing records are still taken into account! If you want to use the new module ID scheme, you have to delete your records file as well.

20.3 Integrating with asset pipelines

To integrate with asset pipelines, you can consider using plugins like [webpack-manifest-plugin](https://www.npmjs.com/package/webpack-manifest-plugin)¹, or [webpack-assets-manifest](https://www.npmjs.com/package/webpack-assets-manifest)². These solutions emit JSON that maps the original asset path to the new one.

20.4 Conclusion

The project has basic caching behavior now. If you try to modify `index.js` or `component.js`, the vendor bundle should remain the same.

To recap:

- Webpack maintains a **runtime** containing information needed to run the application.
- If the runtime manifest changes, the change invalidates the containing bundle.
- Certain plugins allow you to write the runtime to the generated `index.html`. It's also possible to extract the information to a JSON file. The JSON comes in handy with *Server-Side Rendering*.
- **Records** allow you to store module IDs across builds. As a downside, you have to track the records file.

You'll learn to analyze the build in the next chapter as it's essential for understanding and improving your build.

¹<https://www.npmjs.com/package/webpack-manifest-plugin>

²<https://www.npmjs.com/package/webpack-assets-manifest>

21. Build Analysis

Analyzing build statistics is a good step towards understanding webpack better. The available tooling helps to answer the following questions:

- What's the composition of the project bundles?
- What kind of dependencies do project modules have?
- How does the size of the project change over time?
- Which project modules can be safely dropped?
- Which project modules are duplicates?
- Why is a specific module included to the project bundles?

21.1 Configuring webpack

To make webpack emit analysis information, you should set the `--json` flag and pipe the output to a file as follows:

package.json

```
{
  "scripts": {
    "build:stats": "wp --mode production --json > stats.json"
  }
}
```

The above is the basic setup you need, regardless of your webpack configuration. Execute `npm run build:stats` now. After a while you should find *stats.json* at your project root. This file can be pushed through a variety of tools to understand better what's going on.



To capture timing-related information during the build, set `profile` to `true` in webpack configuration.



`0x`¹ can generate a flamegraph of webpack execution to understand where time is spent.

Node API

Stats can be captured through Node. Since stats can contain errors, so it's a good idea to handle that case separately:

```
const webpack = require("webpack");
const config = require("./webpack.config.js")("production");

webpack(config, (err, stats) => {
  if (err) {
    return console.error(err);
  }

  if (stats.hasErrors()) {
    return console.error(stats.toString("errors-only"));
  }

  console.log(stats);
});
```

To detect how webpack configuration is imported, use `if (require.main === module)` kind of check to detect usage through Node. The idea is then to export the configuration (`module.exports = getConfig;`) for `if` and `do` `module.exports = getConfig(mode);` for the `else` clause.

The technique can be valuable if you want to do further processing on stats although often the other solutions are enough.



If you want JSON output from stats, use `stats.toJson()`. To get *verbose* output, use `stats.toJson("verbose")`. It follows all stat options webpack supports.

¹<https://www.npmjs.com/package/0x>



To mimic the `--json` flag, use `console.log(JSON.stringify(stats.toJson(), null, 2));`. The output is formatted to be readable.

webpack-stats-plugin and webpack-bundle-tracker

If you want to manage stats through a plugin, check out [webpack-stats-plugin](#)². It gives you control over the output and lets you transform it before writing. You can use it to exclude specific dependencies from the output.

[webpack-bundle-tracker](#)³ can capture data while webpack is compiling. It uses JSON for this purpose.

21.2 Enabling a performance budget

Webpack allows you to define a **performance budget**. The idea is that it gives your build size constraint, which it has to follow. The feature is disabled by default, and the calculation includes extracted chunks to entry calculation.

To integrate the feature into the project, adjust the configuration as below:

webpack.config.js

```
const productionConfig = merge([
  ...
  {
    performance: {
      hints: "warning", // "error" or false are valid too
      maxEntrypointSize: 50000, // in bytes, default 250k
      maxAssetSize: 100000, // in bytes
    },
  },
]);
```

²<https://www.npmjs.com/package/webpack-stats-plugin>

³<https://www.npmjs.com/package/webpack-bundle-tracker>

In case your project exceeds the limits, you should see a warning similar to below:

```
WARNING in entrypoint size limit: The following entrypoint(s) combined \
asset size exceeds the recommended limit (48.8 KiB). This can impact we\
b performance.
```

Entrypoints:

```
main (131 KiB)
  runtime.41f8.js
  vendor.1622.js
  main.aca1.css
  main.eddd.js
```

If you want to enforce a strict limit in a CI environment, set `hints` to `error`. Doing this will fail the build in case it is reached and force the developers either go below the limit or raise a discussion about good limits.

21.3 Dependency analysis

It's possible to analyze bundle dependencies in a graphical manner, and many tools exist for this purpose:

- [The official analyse tool](https://github.com/webpack/analyse)⁴ gives you recommendations and a good idea of your application's dependency graph. It can be run locally as well.
- [Statoscope](https://statoscope.tech/)⁵ is comparable to the official analyse tool except for the lack of a graph view and it comes with additional filters to understand the output better.
- [circular-dependency-plugin](https://www.npmjs.com/package/circular-dependency-plugin)⁶ lets you detect cycles in the module graph. Often this implies a bug, and it can be a good idea to refactor cycles out.
- [dependency-cruiser](https://www.npmjs.com/package/dependency-cruiser)⁷ is a bundler independent tool for analyzing project dependencies.
- [madge](https://www.npmjs.com/package/madge)⁸ is another independent tool that can output a graph based on module input. The graph output allows you to understand the dependencies of your project in greater detail.

⁴<https://github.com/webpack/analyse>

⁵<https://statoscope.tech/>

⁶<https://www.npmjs.com/package/circular-dependency-plugin>

⁷<https://www.npmjs.com/package/dependency-cruiser>

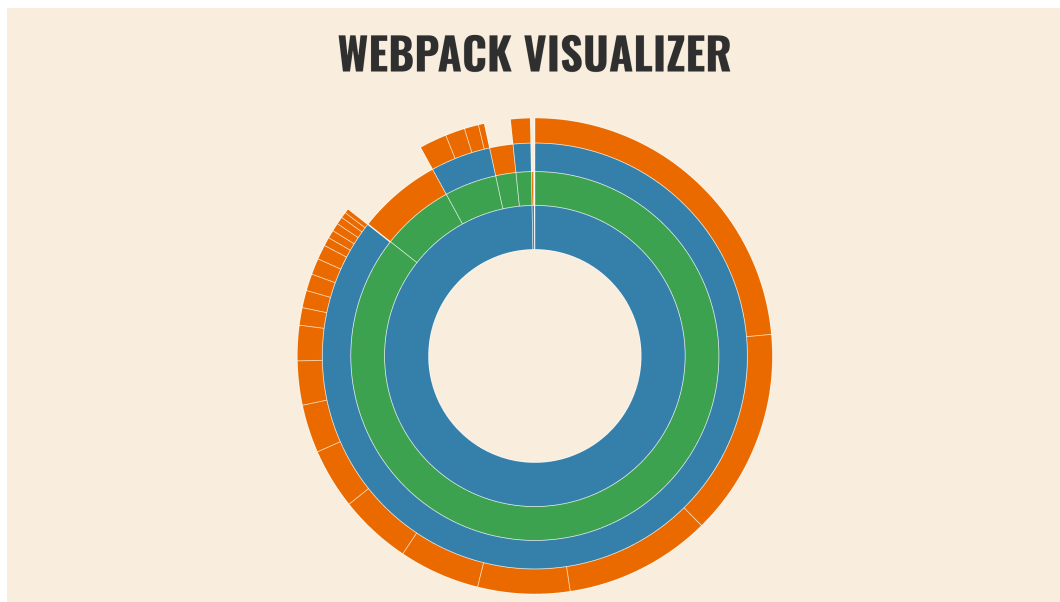
⁸<https://www.npmjs.com/package/madge>

- [Arkit](#)⁹ goes a step beyond madge and it constructs entire architectural overviews of projects.

21.4 Composition analysis

Pie charts, treemaps, and command-line tools let you visualize bundle composition. Studying the generated graphics can generate insights and understand what's contributing to the bundle size.

Pie charts



Webpack Visualizer

[Webpack Visualizer](#)¹⁰ provides a pie chart showing your bundle composition, allowing to understand which dependencies contribute to the size of the overall result. [Webpack Chart](#)¹¹ is another similar option.

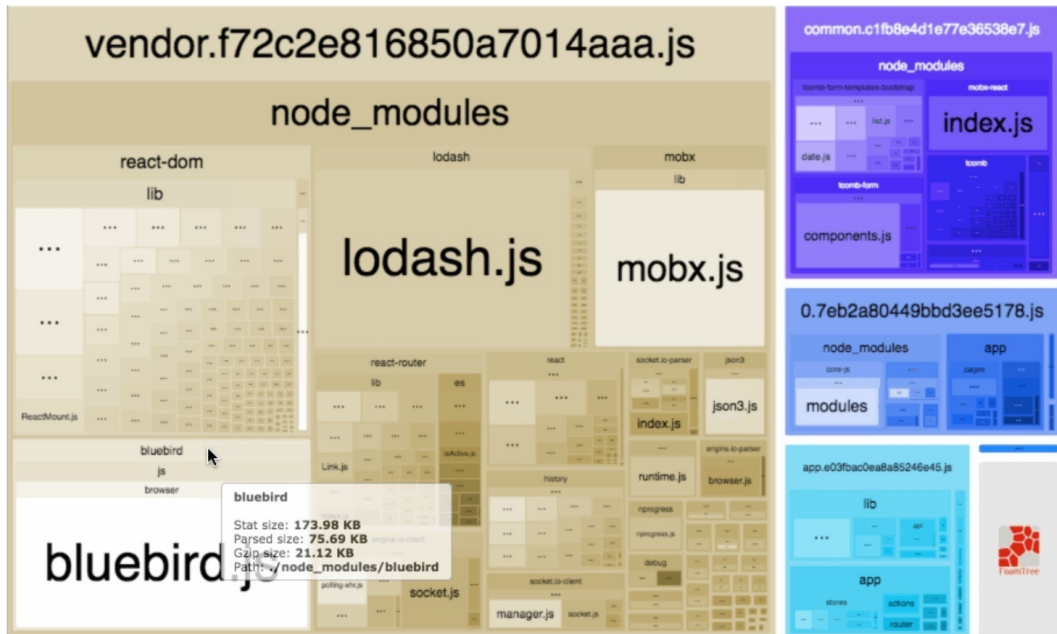
⁹<https://arkit.pro>

¹⁰<https://chrisbateman.github.io/webpack-visualizer/>

¹¹<https://alexkuz.github.io/webpack-chart/>

In addition to providing a pie chart visualization, [Auxpack](http://auxpack.com/)¹² is able to track bundle size over time.

Treemaps



webpack-bundle-analyzer

[webpack-bundle-analyzer](https://www.npmjs.com/package/webpack-bundle-analyzer)¹³ provides a zoomable treemap.

[source-map-explorer](https://www.npmjs.com/package/source-map-explorer)¹⁴ is a tool independent of webpack. It allows you to get insight into your build by using source maps. It gives a treemap based visualization showing what code contributes to the result. [bundle-wizard](https://www.npmjs.com/package/bundle-wizard)¹⁵ is another similar tool.

¹²<http://auxpack.com/>

¹³<https://www.npmjs.com/package/webpack-bundle-analyzer>

¹⁴<https://www.npmjs.com/package/source-map-explorer>

¹⁵<https://www.npmjs.com/package/bundle-wizard>

webpack-bundle-size-analyzer

[webpack-bundle-size-analyzer](#)¹⁶ emits a text based composition:

```
$ webpack-bundle-size-analyzer stats.json
react: 93.99 KB (74.9%)
purecss: 15.56 KB (12.4%)
style-loader: 6.99 KB (5.57%)
fbjs: 5.02 KB (4.00%)
object-assign: 1.95 KB (1.55%)
css-loader: 1.47 KB (1.17%)
<self>: 572 B (0.445%)
```

21.5 Output plugins

There are multiple plugins to make the webpack output easier to understand and more convenient:

- [webpackbar](#)¹⁷ has been made especially for tracking build progress.
- `webpack.ProgressPlugin` is included out of the box and can be used as well.
- [webpack-dashboard](#)¹⁸ gives an entire terminal-based dashboard over the standard webpack output. If you prefer clear visual output, this one comes in handy.

21.6 Online services

It's possible to integrate bundle analysis to your build process by using [Bundle Analyzer](#)¹⁹ (free) and [Packtracker](#)²⁰ (commercial). The services integrate well with GitHub and will show up in your pull requests, as it's valuable to have the information visible there.

¹⁶<https://www.npmjs.com/package/webpack-bundle-size-analyzer>

¹⁷<https://www.npmjs.com/package/webpackbar>

¹⁸<https://www.npmjs.com/package/webpack-dashboard>

¹⁹<https://www.bundle-analyzer.com>

²⁰<https://packtracker.io/>

21.7 Bundle comparison

There are multiple packages which let you compare webpack bundles over time:

- [bundle-stats-webpack-plugin](#)²¹ lets you generate graphical bundle reports and compare them across builds.
- [webpack-bundle-diff](#)²² operates on a lower level and emits a JSON file with the difference. It could work well with a custom visualization.
- [size-plugin](#)²³ prints out the size difference since the last build, and it can be useful during local development.
- [webpack-bundle-delta](#)²⁴ computes a delta between the base branch and the head in pull requests giving you a better idea of the impact of a change size-wise.

21.8 Unused files analysis

[unused-webpack-plugin](#)²⁵ is able to discover files that aren't used by the webpack build but are included to the project. [remnants](#)²⁶ is a solution that goes beyond webpack and can be used with other tools as well.

21.9 Duplication analysis

There are multiple tools for finding duplicates in a project:

- [inspectpack](#)²⁷ has both a command line tool and a webpack plugin for finding duplicate packages.
- [find-duplicate-dependencies](#)²⁸ achieves the same on an npm package level.
- [depcheck](#)²⁹ goes further and warns if there are redundant dependencies or dependencies missing from the project.

²¹<https://www.npmjs.com/package/bundle-stats-webpack-plugin>

²²<https://www.npmjs.com/package/webpack-bundle-diff>

²³<https://www.npmjs.com/package/size-plugin>

²⁴<https://github.com/trainline/webpack-bundle-delta>

²⁵<https://www.npmjs.com/package/unused-webpack-plugin>

²⁶<https://www.npmjs.com/package/remnants>

²⁷<https://www.npmjs.com/package/inspectpack>

²⁸<https://www.npmjs.com/package/find-duplicate-dependencies>

²⁹<https://www.npmjs.com/package/depcheck>

21.10 Understanding why a module was bundled

[whybundled](#)³⁰ has been designed to answer the question why a specific module was included to the bundles. [statoscope](#)³¹ is a visual interface for the same purpose.



Set `stats.reasons` to `true` through webpack configuration to capture similar information.

21.11 Conclusion

When you are optimizing the size of your bundle output, these tools are invaluable. The official tool has the most functionality, but even basic visualization can reveal problem spots. You can use the same technique with old projects to understand their composition.

To recap:

- Webpack allows you to extract a JSON file containing information about the build. The data can include build composition and timing.
- The generated data can be analyzed using various tools that give insight into aspects such as the bundle composition.
- **Performance budget** allows you to set limits to the build size. Maintaining a budget can keep developers more conscious of the size of the generated bundles.
- Understanding the bundles is the key to optimizing the overall size, what to load and when. It can also reveal more significant issues, such as redundant data.

You'll learn to tune webpack performance in the next chapter.

³⁰<https://www.npmjs.com/package/whybundled>

³¹<https://www.npmjs.com/package/@statoscope/ui-webpack>

22. Performance

Webpack's performance out of the box is often enough for small projects. That said, it begins to hit limits as your project grows in scale, and it's a frequent topic in webpack's issue tracker.

There are a couple of ground rules when it comes to optimization:

1. Know what to optimize.
2. Perform fast to implement tweaks first.
3. Perform more involved tweaks after.
4. Measure the impact as you go.

Sometimes optimizations come with a cost. You could, for example, trade memory for performance or end up making your configuration more complicated.



If you hit memory limits with webpack, you can give it more memory with `node --max-old-space-size=4096 node_modules/.bin/wp --mode development` kind of invocation. Size is given in megabytes, and in the example you would give 4 gigabytes of memory to the process.

22.1 Measuring impact

As discussed in the previous chapter, generating stats can be used to measure build time. [webpack.debug.ProfilingPlugin](https://webpack.js.org/plugins/profiling-plugin/)¹ and [cpuprofile-webpack-plugin](https://github.com/jantimon/cpuprofile-webpack-plugin)² are able to emit the timings of plugin execution as a file you can pass to Chrome Inspector. The latter generates a flame graph as well.

¹<https://webpack.js.org/plugins/profiling-plugin/>

²<https://github.com/jantimon/cpuprofile-webpack-plugin>

22.2 High-level optimizations

Webpack uses only a single instance by default, meaning you aren't able to benefit from a multi-core processor without extra effort. This is where solutions like [thread-loader](https://www.npmjs.com/package/thread-loader)³ come in. [webpack-plugin-ramdisk](https://www.npmjs.com/package/webpack-plugin-ramdisk)⁴ writes the build output to a RAM disk and it can help during development and in case you have to perform many successive builds.

22.3 Low-level optimizations

Specific lower-level optimizations can be nice to know. The key is to allow webpack to perform less work. Consider the examples below:

- Use faster source map variants during development or skip them. Skipping is possible if you don't process the code in any way.
- Use [@babel/preset-env](https://www.npmjs.com/package/@babel/preset-env)⁵ to transpile fewer features for modern browsers and make the code more readable and more comfortable to debug while dropping source maps.
- Skip polyfills during development. Attaching a package, such as [core-js](https://www.npmjs.com/package/core-js)⁶, to the development version of an application adds processing overhead.
- Polyfill less of Node and provide nothing instead. For example, a package could be using Node process which in turn will bloat your bundle if polyfilled. See [webpack documentation](https://webpack.js.org/configuration/node/)⁷ for the default values.
- Starting from version 5, there's a [file system level cache](https://webpack.js.org/configuration/cache/)⁸ that can be enabled by setting `cache.type = "filesystem"`. To invalidate it on configuration change, you should set `cache.buildDependencies.config = [__filename]`. Webpack handles anything watched by the build automatically including plugins, loaders, and project files.

³<https://www.npmjs.com/package/thread-loader>

⁴<https://www.npmjs.com/package/webpack-plugin-ramdisk>

⁵<https://www.npmjs.com/package/@babel/preset-env>

⁶<https://www.npmjs.com/package/core-js>

⁷<https://webpack.js.org/configuration/node/>

⁸<https://github.com/webpack/webpack/blob/master/guides/persistent-caching.md>

Loader specific optimizations

Loaders have their optimizations as well:

- Perform less processing by skipping loaders during development. Especially if you are using a modern browser, you can skip using **babel-loader** or equivalent altogether.
- Use either `include` or `exclude` with JavaScript specific loaders. Webpack traverses `node_modules` by default, and executes **babel-loader** over the files unless it has been configured correctly.
- Parallelize the execution of expensive loaders using **thread-loader**. Given workers come with an overhead in Node, the loader is worth it only if the parallelized operation is heavy.

22.4 Optimizing rebundling speed during development

Rebundling times during development can be improved by pointing the development setup to a minified version of a library, such as React. In React's case, you lose `propTypes`-based validation but if speed is paramount, this technique is worth it.

`module.noParse` accepts a `RegExp` or an array of `RegExps`. In addition to telling webpack not to parse the minified file you want to use, you have to point `react` to it by using `resolve.alias`. The idea is discussed in detail in the *Consuming Packages* chapter.

You can encapsulate the idea within a function:

```
exports.dontParse = ({ name, path }) => ({
  module: { noParse: [new RegExp(path)] },
  resolve: { alias: { [name]: path } },
});
```

To use the function, you call it as follows:

```
dontParse({
  name: "react",
  path: path.resolve(
    __dirname, "node_modules/react/cjs/react.production.min.js",
  ),
}),
}),
```

After this change, the application should be faster to rebuild, depending on the underlying implementation. The technique can also be applied to production.

Given `module.noParse` accepts a regular expression if you wanted to ignore all `*.min.js` files, you could set it to `/\..min\.js/`.



Not all modules support `module.noParse`. They should not have a reference to `require`, `define`, or similar, as that leads to an `Uncaught ReferenceError: require is not defined error`.

22.5 Webpack 4 performance tricks

There are various webpack 4 specific tricks to improve performance:

- If `output.futureEmitAssets` is set, webpack 5 related logic is enabled. [Based on Shawn Wang⁹](#), it reduces memory usage and improves situation.
- Sometimes there are version related performance regressions which can be fixed in the user space [Kenneth Chau¹⁰](#) has compiled a great list of them for webpack 4. The main ideas are related to simplifying `stats.toJson` using **ts-loader** with `experimentalWatchApi` and setting `output.pathinfo` to `false`.
- [Jared Palmer mentions¹¹](#) that setting `optimization` property and its `splitChunks`, `removeAvailableModules`, and `removeEmptyChunks` properties to `false` can improve performance in the development mode.

⁹<https://twitter.com/swyx/status/1218173290579136512>

¹⁰https://medium.com/@kenneth_chau/speeding-up-webpack-typescript-incremental-builds-by-7x-3912ba4c1d15

¹¹<https://twitter.com/jaredpalmer/status/1265298834906910729>

22.6 Conclusion

You can optimize webpack's performance in multiple ways. Often it's a good idea to start with more accessible techniques before moving to more involved ones. The exact methods you have to use depend on the project.

To recap:

- Start with high-level techniques that are fast to implement first.
- Lower level techniques are more involved but come with their wins.
- Since webpack runs using a single instance by default, parallelizing is worthwhile.
- Especially during development, skipping work can be acceptable thanks to modern browsers.



The official build performance guide¹² and Web Fundamentals by Google¹³ have more tips.

¹²<https://webpack.js.org/guides/build-performance/>

¹³<https://developers.google.com/web/fundamentals/performance/webpack/>

VI Output

This part covers different output techniques webpack provides. You see how to manage a multi-page setup, how to implement server-side rendering, and how to use module federation to develop micro frontends.

23. Build Targets

Even though webpack is used most commonly for bundling web applications, it can do more. You can use it to target Node or desktop environments, such as Electron. Webpack can also bundle as a library while writing an appropriate output wrapper making it possible to consume the library.

Webpack's output target is controlled by the `target` field. You'll learn about the primary targets next and dig into library-specific options after that.

23.1 Web targets

Webpack uses the `web` target by default. The target is ideal for a web application like the one you have developed in this book. Webpack bootstraps the application and loads its modules. The initial list of modules to load is maintained in a manifest, and then the modules can load each other as defined.

Starting from webpack 5, the default is set to `browserslist` in case a `browserslist` configuration has been found. The change means that webpack will compile its runtime to match the setting instead of generating code that will work in legacy browsers as well. Webpack can target specific language specifications (i.e. `es2020`) and also an array of targets is possible (i.e. `["web", "es2020"]`).

Web workers

The `webworker` target wraps your application as a [web worker](#)¹. Using web workers is valuable if you want to execute computation outside of the main thread of the application without slowing down the user interface. There are a couple of limitations you should be aware of:

- You cannot use webpack's hashing features when the `webworker` target is used.
- You cannot manipulate the DOM from a web worker. If you wrapped the book project as a worker, it would not display anything.



Web workers and their usage are discussed in detail in the *Web Workers* chapter.

23.2 Node targets

Webpack provides two Node-specific targets: `node` and `async-node`. It uses standard Node `require` to load chunks unless the `async` mode is used. In that case, it wraps modules so that they are loaded asynchronously through Node `fs` and `vm` modules.

The main use case for using the Node target is *Server-Side Rendering* (SSR).

Starting from webpack 5, it's possible to target a specific version of Node using for example `node10.13`.



If you develop a server using webpack, see [nodemon-webpack-plugin](#)². The plugin is able to restart your server process without having to set up an external watcher.

¹https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API

²<https://www.npmjs.com/package/nodemon-webpack-plugin>

23.3 Desktop targets

There are desktop shells, such as [NW.js](#)³ (previously **node-webkit**) and [Electron](#)⁴ (previously **Atom**). Webpack can target these as follows:

- `node-webkit` - Targets NW.js while considered experimental.
- `atom`, `electron`, `electron-main` - Targets [Electron main process](#)⁵.
- `electron-renderer` - Targets Electron renderer process.

[electron-react-boilerplate](#)⁶ is a good starting point if you want hot loading webpack setup for Electron and React-based development. Using [the official quick start for Electron](#)⁷ is one way.

23.4 Conclusion

Webpack supports targets beyond the web. Based on this, you can say name “webpack” is an understatement considering its capabilities.

To recap:

- Webpack’s output target can be controlled through the `target` field. It defaults to `web` but accepts other options too.
- Webpack can target the desktop, Node, and web workers in addition to its web target.
- The Node targets come in handy if especially in *Server-Side Rendering* setups.

You’ll learn how to handle multi-page setups in the next chapter.

³<https://nwjs.io/>

⁴<http://electron.atom.io/>

⁵<https://github.com/electron/electron/blob/master/docs/tutorial/quick-start.md>

⁶<https://github.com/electron-react-boilerplate/electron-react-boilerplate>

⁷<https://github.com/electron/electron-quick-start>

24. Multiple Pages

Even though webpack is often used for bundling single-page applications, it's possible to use it with multiple separate pages as well. The idea is similar to the way you generated many output files in the *Targets* chapter. That's achievable through `MiniHtmlWebpackPlugin` and a bit of configuration.



If you want to map a directory tree as a website, see [directory-tree-webpack-plugin¹](#).

24.1 Possible approaches

When generating multiple pages with webpack, you have a couple of possibilities:

- Go through the *multi-compiler mode* and return an array of configurations. The approach would work as long as the pages are separate, and there is a minimal need for sharing code across them.
- Set up a single configuration and extract the commonalities. The way you do this can differ depending on how you chunk it up.
- If you follow the idea of [Progressive Web Applications²](#) (PWA), you can end up with either an **app shell** or a **page shell** and load portions of the application as it's used.

In practice, you have more dimensions. For example, you have to generate 118n variants for pages. These ideas grow on top of the basic approaches. Here we'll set up single configuration based on which to experiment further.

¹<https://www.npmjs.com/package/directory-tree-webpack-plugin>

²<https://developers.google.com/web/progressive-web-apps/>

24.2 Generating multiple pages

To generate multiple pages with webpack, we can leverage **mini-html-webpack-plugin**. **html-webpack-plugin** would work well for the purpose as well and using it would give you access to the plugins written for it. For the demonstration, using the former is enough.

A page should receive title, url, and chunks for deciding which scripts to include to the page. The idea can be modeled as a configuration part as below:

webpack.parts.js

```
const {
  MiniHtmlWebpackPlugin,
} = require("mini-html-webpack-plugin");

exports.page = ({ title, url = "", chunks } = {}) => ({
  plugins: [
    new MiniHtmlWebpackPlugin({
      publicPath: "/",
      chunks,
      filename: `${url && url + "/" }index.html`,
      context: { title },
    }),
  ],
});
```

To generate multiple pages using the new helper, set up a configuration file:

webpack.multi.js

```
const { merge } = require("webpack-merge");
const parts = require("../webpack.parts");

module.exports = merge(
  { mode: "production", entry: { app: "../src/multi.js" } },
  parts.page({ title: "Demo" }),
  parts.page({ title: "Another", url: "another" })
);
```

Implement a small module to render on the page:

src/multi.js

```
const element = document.createElement("div");
element.innerHTML = "hello multi";
document.body.appendChild(element);
```

And add a script to generate the pages:

package.json

```
{
  "scripts": {
    "build:multi": "wp --config webpack.multi.js"
  }
}
```

Testing the build

After these steps, you have a minimal build with two pages: / and /another. To see it in the browser, run `npm run serve` to display the result. You should be able to navigate to both pages any other should not be available.

To control which entries are used on each page, use the `chunks` parameter of `parts.page`. If you set it to `chunks: []` for one of the pages, you should see nothing on the page for example. While experimenting, match the name given at `parts.entry`. The parameter allows capturing chunks generated by *Bundle Splitting* and doing this would allow you to load a shared vendor bundle for all pages.



To support development mode, see the *Composing Configuration* chapter on how to set it up. The development target requires `parts.devServer()` helper and a more complex merge operation based on the target type.

24.3 Progressive web applications

If you push the idea further by combining it with code splitting and smart routing, you'll end up with the idea of Progressive Web Applications (PWA). [webpack-pwa](#)³ example illustrates how to implement the approach using webpack either through an app shell or a page shell.

App shell is loaded initially, and it manages the whole application, including its routing. Page shells are more granular, and more are loaded as the application is used. The total size of the application is larger in this case. Conversely, you can load initial content faster.

Using [Service Workers](#)⁴ improves offline experience and especially [Workbox](#)⁵ and its associated [workbox-webpack-plugin](#)⁶ can be useful for setting up the approach with minimal effort.



[Twitter](#)⁷ and [Tinder](#)⁸ case studies illustrate how the PWA approach can improve platforms.

³<https://github.com/webpack/webpack-pwa>

⁴https://developer.mozilla.org/en/docs/Web/API/Service_Worker_API

⁵<https://developers.google.com/web/tools/workbox/>

⁶<https://www.npmjs.com/package/workbox-webpack-plugin>

⁷<https://developers.google.com/web/showcase/2017/twitter>

⁸<https://medium.com/@addyosmani/a-tinder-progressive-web-app-performance-case-study-78919d98ece0>

24.4 Conclusion

Webpack allows you to manage multiple page setups. The PWA approach allows the application to be loaded as it's used and webpack allows implementing it.

To recap:

- Webpack can be used to generate separate pages either through its multi-compiler mode or by including all the page configuration into one.
- The multi-compiler configuration can run in parallel using external solutions, but it's harder to apply techniques such as bundle splitting against it.
- A multi-page setup can lead to a **Progressive Web Application**. In this case, you use various webpack techniques to come up with an application that is fast to load and that fetches functionality as required. Both two flavors of this technique have their own merits.

You'll learn to implement *Server-Side Rendering* in the next chapter.

25. Server-Side Rendering

Server-Side Rendering (SSR) is a technique that allows you to serve an initial payload with HTML, JavaScript, CSS, and even application state. You serve a fully rendered HTML page that would make sense even without JavaScript enabled. In addition to providing potential performance benefits, this can help with Search Engine Optimization (SEO).

Even though the idea does not sound that unique, there is a technical cost. The approach was popularized by React. Since then frameworks encapsulating the tricky bits, such as [Next.js](https://www.npmjs.com/package/next)¹ and [razzle](https://www.npmjs.com/package/razzle)², have appeared.

To demonstrate SSR, you can use webpack to compile a client-side build that then gets picked up by a server that renders it using React following the principle. Doing this is enough to understand how it works and also where the problems begin.

25.1 Setting up Babel with React

To use React, we require specific configuration. Given most of React projects rely on [JSX](https://facebook.github.io/jsx/)³ format, you have to enable it through Babel:

```
npm add babel-loader @babel/core @babel/preset-react -D
```

¹<https://www.npmjs.com/package/next>

²<https://www.npmjs.com/package/razzle>

³<https://facebook.github.io/jsx/>

Connect the preset with Babel configuration as follows:

.babelrc

```
{
  "presets": [
    ["@babel/preset-env", { "modules": false }],
    "@babel/preset-react"
  ]
}
```

25.2 Setting up a React demo

To make sure the project has the dependencies in place, install React and **react-dom** to render the application to the DOM.

```
npm add react react-dom
```

Next, the React code needs a small entry point. For browser, we'll render a `div` and show an alert on click. For server, we return JSX to render.

As ES2015 style imports and CommonJS exports cannot be mixed, the entry point has to be written in CommonJS style. Adjust as follows:

src/ssr.js

```
const React = require("react");
const ReactDOM = require("react-dom");
const SSR = <div onClick={() => alert("hello")}>Hello world</div>;

// Render only in the browser, export otherwise
if (typeof document === "undefined") {
  module.exports = SSR;
} else {
  ReactDOM.hydrate(SSR, document.getElementById("app"));
}
```


25.3 Configuring webpack

To keep things nice, we will define a separate configuration file. A lot of the work has been done already. Given you have to consume the same output from multiple environments, using UMD as the library target makes sense:

webpack.ssr.js

```
const path = require("path");
const APP_SOURCE = path.join(__dirname, "src");

module.exports = {
  mode: "production",
  entry: { index: path.join(APP_SOURCE, "ssr.js") },
  output: {
    path: path.join(__dirname, "static"),
    filename: "[name].js",
    libraryTarget: "umd",
    globalObject: "this",
  },
  module: {
    rules: [
      {
        test: /\.js$/,
        include: APP_SOURCE,
        use: "babel-loader",
      },
    ],
  },
};
```

To make it convenient to generate a build, add a helper script:

package.json

```
{
  "scripts": {
    "build:ssr": "wp --config webpack.ssr.js"
  }
}
```

If you build the SSR demo (`npm run build:ssr`), you should see a new file at `./static/index.js`. The next step is to set up a server to render it.

25.4 Setting up a server

To keep things clear to understand, you can set up a standalone Express server that picks up the generated bundle and renders it following the SSR principle. Install Express first:

```
npm add express -D
```

Then, to get something running, implement a server:

server.js

```
const express = require("express");
const { renderToString } = require("react-dom/server");
const SSR = require("./static");

const app = express();
app.use(express.static("static"));
app.get("/", (req, res) =>
  res.status(200).send(renderMarkup(renderToString(SSR)))
);
app.listen(parseInt(process.env.PORT, 10) || 8080);
```

```
function renderMarkup(html) {  
  return `<!DOCTYPE html>  
<html>  
  <head><title>SSR Demo</title><meta charset="utf-8" /></head>  
  <body>  
    <div id="app">${html}</div>  
    <script src="./index.js"></script>  
  </body>  
</html>`;   
}
```

Run the server now (`node ./server.js`) and go below `http://localhost:8080`, you should see a “Hello World”. Clicking the text should show an alert and you should see pre-rendered HTML in the source.

Even though there is a React application running now, it’s difficult to develop. If you try to modify the code, nothing happens. The problem can be solved for example by using [webpack-dev-middleware](#)⁴.



If you want to debug output from the server, set `export DEBUG=express:application`.

25.5 Open questions

Even though the demo illustrates the basic idea of SSR, it still leaves open questions:

- How to deal with styles? Node doesn’t understand CSS related imports.
- How to deal with anything other than JavaScript? If the server side is processed through webpack, this is less of an issue as you can patch it at webpack.
- How to run the server through something else other than Node? One option would be to wrap the Node instance in a service you then run through your host environment. Ideally, the results would be cached, and you can find more specific solutions for this particular per platform (i.e. Java and others).

⁴<https://www.npmjs.com/package/webpack-dev-middleware>

Questions like these are the reason why solutions such as Next.js or Razzle exist. They have been designed to solve SSR-specific problems like these.



Webpack provides [require.resolveWeak](https://webpack.js.org/api/module-methods/#requireresolveweak)⁵ for implementing SSR. It's a specific feature used by solutions such as [react-universal-component](https://www.npmjs.com/package/react-universal-component)⁶ underneath.



`__non_webpack_require__(path)` allows you to separate imports that should be evaluated outside of webpack. See the [issue #4175](https://github.com/webpack/webpack/issues/4175)⁷ for more information.

25.6 Prerendering

SSR isn't the only solution to the SEO problem. **Prerendering** is an alternate technique that is easier to implement. The point is to use a headless browser to render the initial HTML markup of the page and then serve that to the crawlers. The caveat is that the approach won't work well with highly dynamic data.

The following solutions exist for webpack:

- [prerender-spa-plugin](https://www.npmjs.com/package/prerender-spa-plugin)⁸ uses [Puppeteer](https://www.npmjs.com/package/puppeteer)⁹ underneath.
- [prerender-loader](https://www.npmjs.com/package/prerender-loader)¹⁰ integrates with **html-webpack-plugin** but also works without it against HTML files. The loader is flexible and can be customized to fit your use case (i.e. React or other framework).

⁵<https://webpack.js.org/api/module-methods/#requireresolveweak>

⁶<https://www.npmjs.com/package/react-universal-component>

⁷<https://github.com/webpack/webpack/issues/4175>

⁸<https://www.npmjs.com/package/prerender-spa-plugin>

⁹<https://www.npmjs.com/package/puppeteer>

¹⁰<https://www.npmjs.com/package/prerender-loader>

25.7 Conclusion

SSR comes with a technical challenge, and for this reason, specific solutions have appeared around it. Webpack is a good fit for SSR setups.

To recap:

- **Server-Side Rendering** (SSR) can provide more for the browser to render initially. Instead of waiting for the JavaScript to load, you can display markup instantly.
- SSR also allows you to pass initial payload of data to the client to avoid unnecessary queries to the server.
- Webpack can manage the client-side portion of the problem. It can be used to generate the server as well if a more integrated solution is required. Abstractions, such as Next.js, hide these details.
- SSR does not come without a cost, and it leads to new problems as you need better approaches for dealing with aspects, such as styling or routing. The server and the client environment differ in essential manners, so the code has to be written so that it does not rely on platform-specific features too much.

In the next chapter, we'll learn about micro frontends and module federation.

26. Module Federation

[Micro frontends](#)¹ take the idea of microservices to frontend development. Instead of developing the application or a site as a monolith, the point is to split it as smaller portions programmed separately that are then tied together during runtime.

With the approach, you can use different technologies to develop other parts of the application and have separate teams developing them. The reasoning is that splitting up development this way avoids the maintenance costs associated with a traditional monolith.

As a side effect, it enables new types of collaboration between backend and frontend developers as they can focus on a specific slice of an application as a cohesive team. For example, you could have a team focusing only on the search functionality or other business-critical portion around a core feature.

Starting from webpack 5, there's built-in functionality to develop micro frontends. **Module federation** and gives you enough functionality to tackle the workflow required by the micro frontend approach.



To learn more about module federation, [see module federation examples](#)² and [Zack Jackson's article about the topic](#)³.

¹<https://micro-frontends.org/>

²<https://github.com/module-federation/module-federation-examples/>

³<https://medium.com/swlh/webpack-5-module-federation-a-game-changer-to-javascript-architecture-bcdd30e02669>

26.1 Module federation example

To get started with module federation, let's build a small application that we'll then split into specific bundles loaded using the technique. The basic requirements of the application are as follows:

1. There should be a UI control with a list of items. Clicking on an item should show related information.
2. There should be a h1 with the application title.
3. From requirement 1., it follows that there should be a main section which will be connected to the control.

Above could be modeled as HTML markup along this:

```
<body>
  <h1>Demo</h1>
  <aside>
    <ul>
      <li><button>Hello world</button></li>
      <li><button>Hello federation</button></li>
      <li><button>Hello webpack</button></li>
    </ul>
  </aside>
  <main>
    The content should change based on what's clicked.
  </main>
</body>
```

The idea is that as any button is clicked, the content is updated to match the text.



To be semantically correct, you could wrap the h1 inside a header.

26.2 Adding webpack configuration

Set up webpack configuration for the project as follows:

webpack.mf.js

```
const path = require("path");
const { mode } = require("webpack-nano/argv");
const { merge } = require("webpack-merge");
const parts = require("./webpack.parts");

const commonConfig = merge([
  {
    entry: [path.join(__dirname, "src", "mf.js")],
    output: { publicPath: "/" },
  },
  parts.loadJavaScript(),
  parts.loadImages(),
  parts.page(),
  parts.extractCSS({ loaders: [parts.tailwind()] }),
]);

const configs = {
  development: merge(
    { entry: ["webpack-plugin-serve/client"] },
    parts.devServer()
  ),
  production: {},
};

module.exports = merge(commonConfig, configs[mode], { mode });
```


The configuration is a subset of what we've used in the book so far. It relies on the following `.babelrc`:

`.babelrc`

```
{
  "presets": [
    "@babel/preset-react",
    ["@babel/preset-env", { "modules": false }]
  ]
}
```

Set up npm scripts as follows:

`package.json`

```
{
  "scripts": {
    "build:mf": "wp --config webpack.mf.js --mode production",
    "start:mf": "wp --config webpack.mf.js --mode development"
  }
}
```

The idea is to have one script to run the project and one to build it.

If you want to improve the setup further, add *Hot Module Replacement* to it, as discussed in the related chapter.



If you haven't completed the book examples, [check out the demonstration from GitHub⁴](https://github.com/survivejs-demos/webpack-demo) to find the configuration.

⁴<https://github.com/survivejs-demos/webpack-demo>

26.3 Implementing the application with React

To avoid manual work with the DOM, we can use React to develop the application quickly. Make sure you have both **react** and **react-dom** installed.

src/mf.js

```
import ReactDOM from "react-dom";
import React from "react";
import "./main.css";

function App() {
  const options = ["Hello world", "Hello fed", "Hello webpack"];
  const [content, setContent] = React.useState("Changes on click.");

  return (
    <main className="max-w-md mx-auto space-y-8">
      <h1 className="text-xl">Demo</h1>
      <aside>
        <ul className="flex space-x-8">
          {options.map((option) => (
            <li key={option}>
              <button
                className="rounded bg-blue-500 text-white p-2"
                onClick={() => setContent(option)}
              >
                {option}
              </button>
            </li>
          ))}
        </ul>
      </aside>
      <article>{content}</article>
    </main>
  );
}
```

```
const container = document.createElement("div");
document.body.appendChild(container);
ReactDOM.render(<App />, container);
```

The styling portion uses Tailwind setup from the *Eliminating Unused CSS* chapter for styling so we can make the demonstration look better.

If you `npm run start:mf`, you should see the application running. In case you click on any of the buttons, the selection should change.

26.4 Separating bootstrap

The next step is breaking the monolith into separate modules. In practice, these portions can be different projects and developed in various technologies.

As a first step, we should use webpack's `ModuleFederationPlugin` and load the application asynchronously. The change in loading is due to the way module federation works. As it's a runtime operation, a small bootstrap is needed.

Add a bootstrap file to the project like this:

src/bootstrap.js

```
import("./mf");
```

It's using the syntax you likely remember from the *Code Splitting* chapter. Although it feels trivial, we need to do this step as otherwise, the application would emit an error while loading with `ModuleFederationPlugin`.

To test the new bootstrap and the plugin, adjust webpack configuration as follows:

```
const { ModuleFederationPlugin } = require("webpack").container;

...

const commonConfig = merge([
  {
    entry: [path.join(__dirname, "src", "mf.js")],
    entry: [path.join(__dirname, "src", "bootstrap.js")],
    output: { publicPath: "/" },
  },
  ...
  {
    plugins: [
      new ModuleFederationPlugin({
        name: "app",
        remotes: {},
        shared: {
          react: { singleton: true },
          "react-dom": { singleton: true },
        },
      }),
    ],
  },
]);

...
```

If you run the application (`npm run start:mf`), it should still look the same.

In case you change the entry to point at the original file, you'll receive an `Uncaught Error: Shared module is not available for eager consumption` error in the browser.

To get started, let's split the header section of the application into a module of its own and load it during runtime through module federation.

Note the singleton bits in the code above. In this case, we'll treat the current code as a host and mark **react** and **react-dom** as a singleton for each federated module to ensure each is using the same version to avoid problems with React rendering.

26.5 Separating header

Now we're in a spot where we can begin breaking the monolith. Set up a file with the header code as follows:

src/header.js

```
import React from "react";

const Header = () => <h1 className="text-xl">Demo</h1>;

export default Header;
```

We should also alter the application to use the new component. We'll go through a custom namespace, **mf**, which we'll manage through module federation:

src/mf.js

```
...

import Header from "mf/header";

function App() {
  ...

  return (
    <main className="max-w-md mx-auto space-y-8">
        <h1 className="text-xl">Demo</h1>
    <Header />
    ...
    </main>
  );
}
```

Next, we should connect the federated module with our configuration. It's here where things get more complicated as we have to either run webpack in multi-compiler mode (array of configurations) or compile modules separately. I've gone with the latter approach, as it works better with the current configuration.



It's possible to make the setup work in a multi-compiler setup as well. In that case, you should either use **webpack-dev-server** or run **webpack-plugin-serve** in a server mode. [See the full example⁵](https://github.com/shellscape/webpack-plugin-serve/blob/master/test/fixtures/multi/webpack.config.js) at their documentation.

To make the changes more manageable, we should define a configuration part encapsulating the module federation concern and then consume that:

webpack.parts.js

```
const { ModuleFederationPlugin } = require("webpack").container;

exports.federateModule = ({
  name,
  filename,
  exposes,
  remotes,
  shared,
}) => ({
  plugins: [
    new ModuleFederationPlugin({
      name,
      filename,
      exposes,
      remotes,
      shared,
    }),
  ],
});
```

⁵<https://github.com/shellscape/webpack-plugin-serve/blob/master/test/fixtures/multi/webpack.config.js>

The next step is more involved, as we'll have to set up two builds. We'll have to reuse the current target and pass `--component` parameter to it to define which one to compile. That gives enough flexibility for the project.

Change the webpack configuration as below:

webpack.mf.js

```
const { mode } = require("webpack-nano/argv");
const { ModuleFederationPlugin } = require("webpack").container;
const { component, mode } = require("webpack-nano/argv");

const commonConfig = merge([
  {
    entry: [path.join(__dirname, "src", "bootstrap.js")],
    output: { publicPath: "/" },
  },
  ...
  parts.extractCSS({ loaders: [parts.tailwind()] }),
  {
    plugins: [
      new ModuleFederationPlugin({
        name: "app",
        remotes: {},
        shared: {
          react: { singleton: true },
          "react-dom": { singleton: true },
        },
      }),
    ],
  },
]);

const shared = {
  react: { singleton: true },
  "react-dom": { singleton: true },
};
```

```

const componentConfigs = {
  app: merge(
    {
      entry: [path.join(__dirname, "src", "bootstrap.js")],
    },
    parts.page(),
    parts.federateModule({
      name: "app",
      remotes: { mf: "mf@mf.js" },
      shared,
    })
  ),
  header: merge(
    {
      entry: [path.join(__dirname, "src", "header.js")],
    },
    parts.federateModule({
      name: "mf",
      filename: "mf.js",
      exposes: { "./header": "./src/header" },
      shared,
    })
  ),
},
};

if (!component) throw new Error("Missing component name");

module.exports = merge(commonConfig, configs[mode], { mode });
module.exports = merge(
  commonConfig,
  configs[mode],
  { mode },
  componentConfigs[component]
);

```


To test, compile the header component first using `npm run build:mf -- --component header`. Then, to run the built module against the shell, use `npm run start:mf -- --component app`.

If everything went well, you should still get the same outcome.

26.6 Pros and cons

You could say our build process is a notch more complex now, so what did we gain? Using the setup, we've essentially split our application into two parts that can be developed independently. The configuration doesn't have to exist in the same repository, and the code could be created using different technologies.

Given module federation is a runtime process, it provides a degree of flexibility that would be hard to achieve otherwise. For example, you could run experiments and see what happens if a piece of functionality is replaced without rebuilding your entire project.

On a team level, the approach lets you have feature teams that work only a specific portion of the application. A monolith may still be a good option for a single developer unless you find the possibility to AB test and to defer compilation valuable.

26.7 Learn more

Consider the following resources to learn more:

- [Module federation at the official documentation](https://webpack.js.org/concepts/module-federation/)⁶
- [module-federation/module-federation-examples](https://github.com/module-federation/module-federation-examples/)⁷
- [mizx/module-federation-examples](https://github.com/mizx/module-federation-examples)⁸
- [Webpack 5 and Module Federation - A Microfrontend Revolution](https://dev.to/marais/webpack-5-and-module-federation-4j1i)⁹
- [The State of Micro Frontends](https://blog.bitsrc.io/state-of-micro-frontends-9c0c604ed13a)¹⁰

⁶<https://webpack.js.org/concepts/module-federation/>

⁷<https://github.com/module-federation/module-federation-examples/>

⁸<https://github.com/mizx/module-federation-examples>

⁹<https://dev.to/marais/webpack-5-and-module-federation-4j1i>

¹⁰<https://blog.bitsrc.io/state-of-micro-frontends-9c0c604ed13a>

26.8 Conclusion

Module federation, introduced in webpack 5, provides an infrastructure-level solution for developing micro frontends.

To recap:

- **Module federation** is a tool-based implementation of micro frontend architecture
- `ModuleFederationPlugin` is the technical implementation of the solution
- When converting a project to use the plugin, set up an asynchronously loaded entry point
- Using the approach brings complexity but at the same time allows you to split your project in ways not possible before

VII Techniques

In this part, you will learn to use webpack techniques such as dynamic loading, using web workers, internationalization, testing, deploying, and package consumption.

27. Dynamic Loading

Even though you can get far with webpack's code splitting features covered in the *Code Splitting* chapter, there's more to it. Webpack provides more dynamic ways to deal with code through `require.context`.

27.1 Dynamic loading with `require.context`

`require.context`¹ provides a general form of code splitting. Let's say you are writing a static site generator on top of webpack. You could model your site contents within a directory structure by having a `./pages/` directory which would contain the 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 and mapped as a site. To model the idea using `require.context`, you could end up with the code as below:

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



The loader definition could be pushed to webpack configuration. The inline form is used to keep the example minimal.

¹<https://webpack.js.org/api/module-methods/#requirecontext>

`require.context` returns 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", body: "# Demo page\nDemo content\n\n"}
const demoPage = req("/demo.md");
```

The technique can be valuable for other purposes, such as testing or adding files for webpack to watch. In that case, you would set up a `require.context` within a file which you then point to through a webpack entry.



If you are using TypeScript, make sure you have installed [@types/webpack-env](https://www.npmjs.com/package/@types/webpack-env)² for `require.context` to work.

27.2 Dynamic paths with a dynamic import

The same idea works with dynamic import. Instead of passing a complete path, you can pass a partial one. Webpack sets up a context internally. Here's a brief example:

```
// Set up a target or derive this somehow
const target = "fi";

// Elsewhere in code
import(`translations/${target}.json`).then(...).catch(...);
```

The same idea works with `require` as webpack can then perform static analysis. For example, `require(assets/modals/${imageSrc}.js)`; would generate a context and resolve against an image based on the `imageSrc` that was passed to the `require`.

²<https://www.npmjs.com/package/@types/webpack-env>



When using dynamic imports, specify file extension in the path as that keeps the context smaller and helps with performance.



There's a full implementation of the idea in the *Internationalization* chapter.

27.3 Combining multiple `require.contexts`

Multiple separate `require.contexts` can be combined into one by wrapping them behind a function:

```
const { concat, uniq } = require("lodash");

const combineContexts = (...contexts) => {
  function webpackContext(req) {
    // Find the first match and execute
    const matches = contexts
      .map((context) => context.keys().indexOf(req) >= 0 && context)
      .filter((a) => a);

    return matches[0] && matches[0](req);
  }
  webpackContext.keys = () =>
    uniq(
      concat.apply(
        null,
        contexts.map((context) => context.keys())
      )
    );
  return webpackContext;
};
```

27.4 Dealing with dynamic paths

Given the approaches discussed here rely on static analysis and webpack has to find the files in question, it doesn't work for every possible case. If the files you need are on another server or have to be accessed through a particular end-point, then webpack isn't enough.

Consider using browser-side loaders like [\\$script.js](https://www.npmjs.com/package/scriptjs)³ or [little-loader](https://www.npmjs.com/package/little-loader)⁴ on top of webpack in this case.

27.5 Conclusion

Even though `require.context` is a niche feature, it's good to be aware of it. It becomes valuable if you have to perform lookups against multiple files available within the file system. If your lookup is more complicated than that, you have to resort to other alternatives that allow you to perform loading runtime.

To recap:

- `require.context` is an advanced feature that's often hidden behind the scenes. Use it if you have to perform a lookup against a large number of files.
- A dynamic `import` written in a certain form generates a `require.context` call. The code reads slightly better in this case.
- The techniques work only against the file system. If you have to operate against urls, you should look into client-side solutions.

The next chapter shows how to use web workers with webpack.

³<https://www.npmjs.com/package/scriptjs>

⁴<https://www.npmjs.com/package/little-loader>

28. Web Workers

[Web workers](#)¹ allow you to push work outside of main execution thread of JavaScript, making them convenient for lengthy computations and background work.

Moving data between the main thread and the worker comes with communication-related overhead. The split provides isolation that forces workers to focus on logic only as they cannot manipulate the user interface directly.

As discussed in the *Build Targets* chapter, webpack allows you to build your application as a worker itself. To get the idea of web workers better, we'll write a small worker to bundle using webpack.

28.1 Setting up a worker

A worker has to do two things: listen to messages and respond. Between those two actions, it can perform a computation. In this case, you accept text data, append it to itself, and send the result:

src/worker.js

```
self.onmessage = ({ data: { text } }) => {  
  self.postMessage({ text: text + text });  
};
```

¹https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API

28.2 Setting up a host

The host has to instantiate the worker and then communicate with it:

src/component.js

```
export default (text = HELLO) => {
  const element = document.createElement("h1");
  const worker = new Worker(
    new URL("./worker.js", import.meta.url)
  );
  const state = { text };

  worker.addEventListener("message", ({ data: { text } }) => {
    state.text = text;
    element.innerHTML = text;
  });
  element.innerHTML = state.text;
  element.onclick = () => worker.postMessage({ text: state.text });

  return element;
};
```

After you have these two set up, it should work as webpack detects the `Worker` syntax. As you click the text, it should mutate the application state when the worker completes its execution. To demonstrate the asynchronous nature of workers, you could try adding delay to the answer and see what happens.

28.3 Sharing data

Due to the cost of serialization, passing data between the host and the worker can be expensive. The cost can be minimized by using [Transferable objects](https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers#Passing_data_by_transferring_ownership)² and in the future, sharing data will become possible thanks to [SharedArrayBuffer](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/SharedArrayBuffer)³.

²https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers#Passing_data_by_transferring_ownership

³https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/SharedArrayBuffer

28.4 Other options

Before webpack 5, [worker-loader](https://www.npmjs.com/package/worker-loader)⁴ was the preferred option and it can still be used if you want more control over the bundling process.

[workerize-loader](https://www.npmjs.com/package/workerize-loader)⁵ and [worker-plugin](https://www.npmjs.com/package/worker-plugin)⁶ let you use the worker as a regular JavaScript module as well given you avoid the `self` requirement visible in the example solution.

[threads.js](https://threads.js.org/)⁷ provides a comprehensive solution for more complex setups and it includes features such as observables and thread pools out of the box.

28.5 Conclusion

The critical thing to note is that the worker cannot access the DOM. You can perform computation and queries in a worker, but it cannot manipulate the user interface directly.

To recap:

- Web workers allow you to push work out of the main thread of the browser. This separation is valuable, especially if performance is an issue.
- Web workers cannot manipulate the DOM. Instead, it's best to use them for lengthy computations and requests.
- The isolation provided by web workers can be used for architectural benefit. It forces the programmers to stay within a specific sandbox.
- Communicating with web workers comes with an overhead that makes them less practical. As the specification evolves, this can change in the future.

You'll learn about internationalization in the next chapter.

⁴<https://www.npmjs.com/package/worker-loader>

⁵<https://www.npmjs.com/package/workerize-loader>

⁶<https://www.npmjs.com/package/worker-plugin>

⁷<https://threads.js.org/>

29. Internationalization

Internationalization (i18n) is a big topic by itself. The broadest definition has to do with translating your user interface to other languages. **Localization** (l10n) is a more specific term, and it describes how to adapt your application to a particular locale or market. Different locales can have the same language, but they still have their customs, like date formatting or measures.

The problem could be solved by pushing the translations behind an endpoint and loading them dynamically to decouple the issue from webpack. Doing this would also allow you to implement a translation interface within your application to enable your translators, or even users, to translate the application. The downside of this approach is that then you have a translation backend to maintain.

Another approach is to let webpack generate static builds, each per language. The problem is that you have to update your application each time your translations change.



See the [Intl JavaScript API](#)¹ to find out what utilities the browsers provide to help with the problem.

29.1 i18n with webpack

The basic idea of i18n with webpack is often the same. You have a translation definition that is then mapped to the application through replacements. The result contains a translated version of the application.

You can use [po-loader](#)² to map [GNU gettext PO files](#)³ to multiple formats, including raw JSON and [Jed](#)⁴.

¹https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Intl

²<https://www.npmjs.com/package/po-loader>

³https://www.gnu.org/software/gettext/manual/html_node/PO-Files.html

⁴<https://messageformat.github.io/Jed/>

Another way is to use webpack's `import()` syntax and *Dynamic Loading* to set up a small system of your own. That's what we'll do next.

29.2 Setting up translations

Set up initial translations as below:

translations/en.json

```
{ "hello": "Hello world" }
```

translations/fi.json

```
{ "hello": "Terve maailma" }
```

29.3 Setting up webpack

If you've implemented webpack configuration in the book so far, you can reuse most of that. The configuration below works standalone and provides a build for React:

webpack.i18n.js

```
const path = require("path");
const {
  MiniHtmlWebpackPlugin,
} = require("mini-html-webpack-plugin");
const APP_SOURCE = path.join(__dirname, "src");

module.exports = {
  mode: "production",
  entry: { index: path.join(APP_SOURCE, "i18n.js") },
  module: {
    rules: [
      {
```

```
      test: /\.js$/,
      include: APP_SOURCE,
      use: "babel-loader",
    },
  ],
},
plugins: [new MiniHtmlWebpackPlugin()],
};
```

The Babel configuration required looks like this:

```
{
  "presets": [
    ["@babel/preset-env", { "modules": false }],
    "@babel/preset-react"
  ]
}
```

To make it convenient to generate the demo application, set up a shortcut:

package.json

```
{
  "scripts": {
    "build:i18n": "wp --config webpack.i18n.js",
  }
}
```

29.4 Setting up application

The last step is to create a small application to load the translations using React and

`import()`:

`src/i18n.js`

```
import "regenerator-runtime/runtime";
import React, { useEffect, useState } from "react";
import ReactDOM from "react-dom";

const App = () => {
  const [language, setLanguage] = useState("en");
  const [hello, setHello] = useState("");

  const changeLanguage = () =>
    setLanguage(language === "en" ? "fi" : "en");

  useEffect(() => {
    translate(language, "hello")
      .then(setHello)
      .catch(console.error);
  }, [language]);

  return (
    <div>
      <button onClick={changeLanguage}>Change language</button>
      <div>{hello}</div>
    </div>
  );
};

function translate(locale, text) {
  return getLocaleData(locale).then((messages) => messages[text]);
}
```

```
async function getLocaleData(locale) {  
  return import(`../messages/${locale}.json`);  
}  
  
const root = document.createElement("div");  
  
root.setAttribute("id", "app");  
document.body.appendChild(root);  
  
ReactDOM.render(<App />, root);
```

If you build (`npm run build:i18n`) and run (`npx serve dist`) the application, you should see that it's loading the translation dynamically and as you click the button, it's changing the translation.



To eliminate that `regenerator-runtime/runtime` import, use Babel's `useBuiltIns` option. It's explained in more detail at the *Loading JavaScript* chapter.

29.5 Conclusion

An internationalization and localization approach can be built on top of webpack. Specific loaders can help in the task as you can push tasks like processing gettext PO files to them.

To recap:

- Webpack supports multiple approaches to i18n and l10n. As a starting point, you can develop a small setup on top of webpack's `import()` syntax.
- A part of the logic can be pushed to loaders for processing PO files for example.

The next chapter covers various testing setups and tools that work with webpack.

30. Testing

Testing is a vital part of development. Even though techniques, such as linting, can help to spot and solve issues, they have their limitations. Testing can be applied to the code and an application on many different levels.

You can **unit test** a specific piece of code, or you can look at the application from the user's point of view through **acceptance testing**. **Integration testing** fits between these ends of the spectrum and is concerned about how separate units of code operate together.

Often you won't need webpack to run your tests. Tools such as [Jest](#)¹, [Cypress](#)², [Puppeteer](#)³, and [Playwright](#)⁴ cover the problem well. Often there are ways to adapt to webpack specific syntax in case you are using webpack features within your code.

30.1 Jest

Facebook's [Jest](#)⁵ is an opinionated alternative that encapsulates functionality, including coverage and mocking, with minimal setup. It can capture snapshots of data making it valuable for projects where you have the behavior you would like to record and retain.

Jest follows [Jasmine](#)⁶ test framework semantics, and it supports Jasmine-style assertions out of the box. Especially the suite definition is close enough to Mocha so that the current test should work without any adjustments to the test code itself. Jest provides [jest-codemods](#)⁷ for migrating more complicated projects to Jest semantics.

¹<https://jestjs.io/>

²<https://www.cypress.io/>

³<https://pptr.dev/>

⁴<https://playwright.dev/>

⁵<https://facebook.github.io/jest/>

⁶<https://www.npmjs.com/package/jasmine>

⁷<https://www.npmjs.com/package/jest-codemods>

Jest captures tests through `package.json` [configuration](#)⁸. It detects tests within a **tests** directory automatically. To capture test coverage information, you have to set `"collectCoverage": true` at "jest" settings in `package.json` or pass `--coverage` flag to Jest. It emits the coverage reports below *coverage* directory by default.

Porting a webpack setup to Jest requires more effort especially if you rely on webpack specific features. [The official guide](#)⁹ covers quite a few of the common problems. You can configure Jest to use Babel through [babel-jest](#)¹⁰ as it allows you to use Babel plugins like [babel-plugin-module-resolver](#)¹¹ to match webpack's functionality.

30.2 Mocking

Mocking is a technique that allows you to replace test objects. Consider using [Sinon](#)¹² for this purpose as it works well with webpack.

30.3 Removing files from tests

If you execute tests through webpack, you may want to alter the way it treats assets like images. You can match them and then use a `noop` function to replace the modules as follows:

```
const config = {
  plugins: [
    new webpack.NormalModuleReplacementPlugin(
      /\.?(gif|png|scss|css)$/ ,
      "lodash/noop"
    ),
  ],
};
```

⁸<https://facebook.github.io/jest/docs/en/configuration.html>

⁹<https://jestjs.io/docs/webpack>

¹⁰<https://www.npmjs.com/package/babel-jest>

¹¹<https://www.npmjs.com/package/babel-plugin-module-resolver>

¹²<https://www.npmjs.com/package/sinon>

30.4 Conclusion

Webpack can be configured to work with a large variety of testing tools. Each tool has its sweet spots, but they also have quite a bit of common ground.

To recap:

- Running testing tools allows you to benefit from webpack's module resolution mechanism.
- Sometimes the test setup can be quite involved. Tools like Jest remove most of the boilerplate and allow you to develop tests with minimal configuration.
- You can find multiple mocking tools for webpack. They allow you to shape test environment. Sometimes you can avoid mocking through design, though.

You'll learn to deploy applications using webpack in the next chapter.

31. Deploying Applications

A project built with webpack can be deployed to a variety of environments. A public project that doesn't rely on a backend can be pushed to GitHub Pages using the **gh-pages** package. Also, there are a variety of webpack plugins that can target other environments, such as S3.

31.1 Deploying with gh-pages

[gh-pages](https://www.npmjs.com/package/gh-pages)¹ allows you to host stand-alone applications on GitHub Pages easily. It has to be pointed to a build directory first. It picks up the contents and pushes them to the gh-pages branch.

Despite its name, the package works with other services that support hosting from a Git repository as well. But given GitHub is so popular, it can be used to demonstrate the idea. In practice, you would likely have more complicated setup in place that would push the result to another service through a Continuous Integration system.

Setting up gh-pages

To get started, execute

```
npm add gh-pages -D
```

¹<https://www.npmjs.com/package/gh-pages>

You are also going to need a script in `package.json`:

`package.json`

```
{
  "scripts": {
    "deploy": "gh-pages -d dist"
  }
}
```

To make the asset paths work on GitHub Pages, `output.publicPath` field has to be adjusted. Otherwise, the asset paths end up pointing at the root, and that doesn't work unless you are hosting behind a domain root (say `survivejs.com`) directly.

`publicPath` gives 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 as below:

`webpack.config.js`

```
const productionConfig = merge([
  {
    output: {
      publicPath: "/",
      // Tweak this to match your GitHub project name
      publicPath: "/webpack-demo/",
    },
  },
  ...
]);
```

After building (`npm run build`) and deploying (`npm run deploy`), you should have your application from the `dist/` directory hosted on GitHub Pages. You should find it at `https://<name>.github.io/<project>` assuming everything went fine.



If you need a more elaborate setup, use the Node API that **gh-pages** provides. The default command line tool it gives is enough for essential purposes, though.



GitHub Pages allows you to choose the branch where you deploy. It's possible to use the `master` branch even as it's enough for minimal sites that don't need bundling. You can also point below the `./docs` directory within your `master` branch and maintain your site.

Archiving old versions

`gh-pages` provides an `add` option for archival purposes. The idea goes as follows:

1. Copy the old version of the site in a temporary directory and remove *archive* directory from it. You can name the archival directory as you want.
2. Clean and build the project.
3. Copy the old version below `dist/archive/<version>`
4. Set up a script to call `gh-pages` through Node as below and capture possible errors in the callback:

```
ghpages.publish(path.join(__dirname, "dist"), { add: true }, cb);
```

31.2 Deploying to other environments

Even though you can push the problem of deployment outside of webpack, there are a couple of webpack specific utilities that come in handy:

- [webpack-deploy](https://www.npmjs.com/package/webpack-deploy)² is a collection of deployment utilities and works even outside of webpack.
- [webpack-s3-plugin](https://www.npmjs.com/package/webpack-s3-plugin)³ sync the assets to Amazon.
- [ssh-webpack-plugin](https://www.npmjs.com/package/ssh-webpack-plugin)⁴ has been designed for deployments over SSH.

²<https://www.npmjs.com/package/webpack-deploy>

³<https://www.npmjs.com/package/webpack-s3-plugin>

⁴<https://www.npmjs.com/package/ssh-webpack-plugin>



To get access to the generated files and their paths, consider using [assets-webpack-plugin](#)⁵. The path information allows you to integrate webpack with other environments while deploying.



To make sure clients relying on the older bundles still work after deploying a new version, do **not** remove the old files until they are old enough. You can perform a specific check on what to remove when deploying instead of removing every old asset.

31.3 Resolving `output.publicPath` dynamically

If you don't know `publicPath` beforehand, it's possible to resolve it based on the environment by following these steps:

1. Set `__webpack_public_path__ = window.myDynamicPublicPath;` in the application entry point and resolve it as you see fit.
2. Remove `output.publicPath` setting from your webpack configuration.
3. If you are using ESLint, set it to ignore the global through `globals.__webpack_public_path__: true`.

When you compile, webpack picks up `__webpack_public_path__` and rewrites it so that it points to webpack logic.



See [webpack documentation](#)⁶ for other webpack specific variables available at the module level.

⁵<https://www.npmjs.com/package/assets-webpack-plugin>

⁶<https://webpack.js.org/api/module-variables/>

31.4 Conclusion

Even though webpack isn't a deployment tool, you can find plugins for it.

To recap:

- It's possible to handle the problem of deployment outside of webpack. You can achieve this in an npm script for example.
- You can configure webpack's `output.publicPath` dynamically. This technique is valuable if you don't know it compile-time and want to decide it later. This is possible through the `__webpack_public_path__` global.

32. Consuming Packages

Sometimes packages have not been packaged the way you expect, and you have to tweak the way webpack interprets them. Webpack provides multiple ways to achieve this.

32.1 `resolve.alias`

Sometimes packages do not follow the standard rules and their `package.json` contains a faulty `main` field. It can be missing altogether. `resolve.alias` is the field to use here as in the example below:

```
const config = {
  resolve: {
    alias: {
      demo: path.resolve(
        __dirname,
        "node_modules/demo/dist/demo.js"
      ),
    },
  },
};
```

The idea is that if webpack resolver matches `demo` in the beginning, it resolves from the target. You can constrain the process to an exact name by using a pattern like `demo$`.

Light React alternatives, such as [Preact](https://www.npmjs.com/package/preact)¹ or [Inferno](https://www.npmjs.com/package/inferno)², offer smaller size while trading off functionality like `propTypes` and synthetic event handling. Replacing React with

¹<https://www.npmjs.com/package/preact>

²<https://www.npmjs.com/package/inferno>

a lighter alternative can save a significant amount of space, but you should test well if you do this.



The same technique works with loaders too. You can use `resolveLoader.alias` similarly. You can use the method to adapt a RequireJS project to work with webpack.

32.2 `resolve.modules`

The module resolution process can be altered by changing where webpack looks for modules. By default, it will look only within the `node_modules` directory. If you want to override packages there, you could tell webpack to look into other directories first:

```
const config = { resolve: { modules: ["demo", "node_modules"] } };
```

After the change, webpack will try to look into the *my_modules* directory first. The method can be applicable in large projects where you want to customize behavior.

32.3 `resolve.extensions`

By default, webpack will resolve only against `.js`, `.mjs`, and `.json` files while importing without an extension, to tune this to include JSX files, adjust as below:

```
const config = { resolve: { extensions: [".js", ".jsx"] } };
```

32.4 `resolve.plugins`

`resolve.plugins` field allows you to customize the way webpack resolves modules. [directory-named-webpack-plugin](https://www.npmjs.com/package/directory-named-webpack-plugin)³ is a good example as it's mapping `import foo from './foo';` to `import foo from './foo/foo.js';`. The pattern is popular with React and using the plugin will allow you to simplify your code. [babel-plugin-module-resolver](https://www.npmjs.com/package/babel-plugin-module-resolver)⁴ achieves the same behavior through Babel.

³<https://www.npmjs.com/package/directory-named-webpack-plugin>

⁴<https://www.npmjs.com/package/babel-plugin-module-resolver>

32.5 Consuming packages outside of webpack

Browser dependencies, like jQuery, are often served through publicly available Content Delivery Networks (CDN). CDNs allow you to push the problem of loading popular packages elsewhere. If a package has been already loaded from a CDN and it's in the user cache, there is no need to load it.

To use this technique, you should first mark the dependency in question as an external:

```
const config = { externals: { jquery: "jquery" } };
```

You still have to point to a CDN and ideally provide a local fallback, so there is something to load if the CDN does not work for the client:

```
<script src="//ajax.googleapis.com/ajax/libs/jquery/3.1.1/jquery.min.js\"
"></script>
<script>
  window.jQuery ||
    document.write(
      '<script src="js/jquery-3.1.1.min.js"></script>'
    );
</script>
```



Starting from webpack 5, the tool supports `externalsType`⁵ field to customize the loading behavior. For example, using "promise" string as its value would load the externals asynchronously and "import" would use browser `import()` to load the externals. This can be configured per external as well instead of using a global setting. To load jQuery asynchronously, you would set it to `["jquery", "promise"]` in the example above.

⁵<https://webpack.js.org/configuration/externals/#externalstype>

32.6 Dealing with globals

Sometimes modules depend on globals. \$ provided by jQuery is a good example. Webpack offers a few ways that allow you to handle them.

Injecting globals

[imports-loader](https://www.npmjs.com/package/imports-loader)⁶ allows you to inject globals to modules. In the example below, import \$ from 'jquery'; is injected as a global to each:

```
const config = {
  module: {
    rules: [
      {
        test: /\.js$/,
        loader: "imports-loader",
        options: {
          imports: ["default jquery $"],
        },
      },
    ],
  },
};
```

Resolving globals

Webpack's ProvidePlugin allows webpack to resolve globals as it encounters them:

```
const config = {
  plugins: [new webpack.ProvidePlugin({ $: "jquery" })],
};
```

⁶<https://www.npmjs.com/package/imports-loader>

Exposing globals to the browser

Sometimes you have to expose packages to third-party scripts. [expose-loader](#)⁷ allows this as follows:

```
const config = {
  test: require.resolve("react"),
  loader: "expose-loader",
  options: {
    exposes: ["React"],
  },
};
```



[script-loader](#)⁸ allows you to execute scripts in a global context. You have to do this if the scripts you are using rely on a global registration setup.

32.7 Managing symbolic links

Symbolic links, or symlinks, are an operating system level feature that allows you to point to other files through a file system without copying them. You can use `npm link` to create global symlinks for packages under development and then use `npm unlink` to remove the links.

Webpack resolves symlinks to their full path as Node does. The problem is that if you are unaware of this fact, the behavior can surprise you especially if you rely on webpack processing. It's possible to work around the behavior as discussed in [webpack issue #985](#)⁹. Webpack core behavior may improve in the future to make a workaround unnecessary. You can disable webpack's symlink handling by setting `resolve.symlinks` as `false`.

⁷<https://www.npmjs.com/package/expose-loader>

⁸<https://www.npmjs.com/package/script-loader>

⁹<https://github.com/webpack/webpack/issues/985>

32.8 Removing unused modules

Even though packages can work well out of the box, they bring too much code to your project sometimes. [Moment.js¹⁰](https://www.npmjs.com/package/moment) is a popular example. It brings locale data to your project by default.

The easiest method to disable that behavior is to use `IgnorePlugin` to ignore locales:

```
const config = {
  plugins: [
    new webpack.IgnorePlugin({
      resourceRegExp: /^\.\/locale$/,
      contextRegExp: /moment$/,
    }),
  ],
};
```



You can use the same mechanism to work around problematic dependencies. Example: `new webpack.IgnorePlugin({ resourceRegExp: /^(buffertools)$/ })`.

To bring specific locales to your project, you should use `ContextReplacementPlugin`:

```
const config = {
  plugins: [
    new webpack.ContextReplacementPlugin(
      /moment[\\\/]locale$/,
      /de|fi/
    ),
  ],
};
```

¹⁰<https://www.npmjs.com/package/moment>



There's a [Stack Overflow question](#)¹¹ that covers these ideas in detail. See also [Ivan Akulov's explanation of ContextReplacementPlugin](#)¹².



You can load locales of `date-fns`¹³ with a similar technique to avoid bundling each.

32.9 Managing pre-built dependencies

It's possible webpack gives the following warning with certain dependencies:

```
WARNING in ../jasmine-promises/dist/jasmine-promises.js
Critical dependencies:
1:113-120 This seems to be a pre-built javascript file. Though this is \
possible, it's not recommended. Try to require the original source to g\
et better results.
@ ../jasmine-promises/dist/jasmine-promises.js 1:113-120
```

The warning can happen if a package points at a pre-built (i.e., minified and already processed) file. Webpack detects this case and warns against it.

The warning can be eliminated by aliasing the package to a source version as discussed above. Given sometimes the source is not available, another option is to tell webpack to skip parsing the files through `module.noParse`. It accepts either a RegExp or an array of RegExps and can be configured as below:

¹¹<https://stackoverflow.com/questions/25384360/how-to-prevent-moment-js-from-loading-locales-with-webpack/25426019>

¹²<https://iamakulov.com/notes/webpack-contextreplacementplugin/>

¹³<https://github.com/date-fns/date-fns/blob/main/docs/webpack.md>

```
const config = {  
  module: { noParse: /node_modules\/demo\/index.js/ },  
};
```



Take care when disabling warnings as it can hide underlying issues. Consider alternatives first. There's a [webpack issue¹⁴](#) that discusses the problem.

32.10 Getting insights on packages

To get more information, npm provides `npm info <package>` command for basic queries. You can use it to check the metadata associated with packages while figuring out version related information.

32.11 Conclusion

Webpack can consume most npm packages without a problem. Sometimes, though, patching is required using webpack's resolution mechanism.

To recap:

- Use webpack's module resolution to your benefit. Sometimes you can work around issues by tweaking resolution. Often it's a good idea to try to push improvements upstream to the projects themselves, though.
- Webpack allows you to patch resolved modules. Given specific dependencies expect globals, you can inject them. You can also expose modules as globals as this is necessary for certain development tooling to work.

¹⁴<https://github.com/webpack/webpack/issues/1617>

VIII Extending

Even though there are a lot of available loaders and plugins for webpack, it's good to be able to extend it. In this part, you go through a couple of short examples to understand how to get started.

33. Extending with Loaders

As you have seen so far, loaders are one of the building blocks of webpack. If you want to load an asset, you most likely need to set up a matching loader definition. Even though there are a lot of [available loaders](https://webpack.js.org/loaders/)¹, it's possible you are missing one fitting your purposes.

You'll learn to develop a couple of small loaders next. But before that, it's good to understand how to debug them in isolation.

33.1 Debugging loaders with loader-runner

[loader-runner](https://www.npmjs.com/package/loader-runner)² allows you to run loaders without webpack, allowing you to learn more about loader development. Install it first:

```
npm add loader-runner -D
```

To have something to test, set up a loader that returns twice what's passed to it:

loaders/demo-loader.js

```
module.exports = (input) => input + input;
```

Create a `demo.txt` file with text in it to the project root as well.

¹<https://webpack.js.org/loaders/>

²<https://www.npmjs.com/package/loader-runner>

There's nothing webpack specific in the code yet. The next step is to run the loader through **loader-runner**:

run-loader.js

```
const fs = require("fs");
const path = require("path");
const { runLoaders } = require("loader-runner");

runLoaders(
  {
    resource: "./demo.txt",
    loaders: [path.resolve(__dirname, "./loaders/demo-loader")],
    readResource: fs.readFile.bind(fs),
  },
  (err, result) => (err ? console.error(err) : console.log(result))
);
```

If you run the script now (`node ./run-loader.js`), you should see output:

```
{
  result: [ 'foobar\nfoobar\n' ],
  resourceBuffer: <Buffer 66 6f 6f 62 61 72 0a>,
  cacheable: true,
  fileDependencies: [ './demo.txt' ],
  contextDependencies: [],
  missingDependencies: []
}
```

The output tells the result of the processing, the resource that was processed as a buffer, and other meta information. The data is enough to develop more complicated loaders.



If you want to capture the output to a file, use either `fs.writeFileSync("./output.txt", result.result)` or its asynchronous version as discussed in [Node documentation](https://nodejs.org/api/fs.html)³.

³<https://nodejs.org/api/fs.html>



It's possible to refer to loaders installed to the local project by name instead of resolving a full path to them. Example: `loaders: ["babel-loader"]`.

33.2 Implementing an asynchronous loader

Even though you can implement a lot of loaders using the synchronous interface, there are times when an asynchronous calculation is required. Wrapping a third-party package as a loader can force you to this.

The example above can be adapted to asynchronous form by using webpack specific API through `this.async()`. Webpack sets this, and the function returns a callback following Node conventions (error first, result second).

Tweak as follows:

loaders/demo-loader.js

```
module.exports = function (input) {  
  const callback = this.async();  
  
  // No callback -> return synchronous results  
  // if (callback) { ... }  
  
  callback(null, input + input);  
};
```



Given webpack injects its API through `this`, the shorter function form (`(() => ...)`) cannot be used here.



If you want to pass a source map to webpack, give it as the third parameter of the callback.

Running the demo script (`node ./run-loader.js`) again should give the same result as before.

To raise an error during execution, try the following:

loaders/demo-loader.js

```
module.exports = function (input) {  
  const callback = this.async();  
  
  callback(new Error("Demo error"));  
};
```

The result should contain `Error: Demo error` with a stack trace showing where the error originates.

33.3 Returning only output

Loaders can be used to output code alone. You could have an implementation as below:

loaders/demo-loader.js

```
module.exports = () => "foobar";
```

But what's the point? You can pass to loaders through webpack entries. Instead of pointing to pre-existing files as you would in a majority of the cases, you could give to a loader that generates code dynamically.



If you want to return Buffer output, set `module.exports.raw = true`. The flag overrides the default behavior, which expects a string is returned.

33.4 Writing files

Loaders, like **file-loader**, emit files. Webpack provides a single method, `this.emitFile`, for this. Given **loader-runner** does not implement it, you have to mock it:

run-loader.js

```
runLoaders(  
  {  
    resource: "./demo.txt",  
    loaders: [path.resolve(__dirname, "./loaders/demo-loader")],  
    context: { emitFile: () => {} },  
    readResource: fs.readFile.bind(fs),  
  },  
  (err, result) => (err ? console.error(err) : console.log(result))  
);
```

To implement the essential idea of asset loading, you have to do two things: emit the file and return path to it.

To interpolate the file name, you need to use [loader-utils](https://www.npmjs.com/package/loader-utils)⁴. Install it first:

```
npm add loader-utils -D
```

⁴<https://www.npmjs.com/package/loader-utils>

Apply the logic as below:

loaders/demo-loader.js

```
const loaderUtils = require("loader-utils");

module.exports = function (content) {
  const url = loaderUtils.interpolateName(this, "[hash].[ext]", {
    content,
  });

  this.emitFile(url, content);

  const path = `__webpack_public_path__ + ${JSON.stringify(url)}`;

  return `export default ${path}`;
};
```

Webpack provides two additional emit methods:

- `this.emitWarning(<string>)`
- `this.emitError(<string>)`

These calls should be used over console based alternatives. As with `this.emitFile`, you have to mock them for **loader-runner** to work.

The next question is how to pass a file name to the loader.

33.5 Passing options to loaders

To demonstrate passing options, the runner needs a small tweak:

run-loader.js

```
const fs = require("fs");
const path = require("path");
const { runLoaders } = require("loader-runner");

runLoaders(
  {
    resource: "./demo.txt",
    loaders: [path.resolve(__dirname, "../loaders/demo-loader")],
    loaders: [
      {
        loader: path.resolve(__dirname, "../loaders/demo-loader"),
        options: {
          name: "demo.[ext]",
        },
      },
    ],
    context: {
      emitFile: () => {},
    },
    readResource: fs.readFile.bind(fs),
  },
  (err, result) => (err ? console.error(err) : console.log(result))
);
```

To connect it to the loader, set it to capture name and pass it through webpack's interpolator:

loaders/demo-loader.js

```
const loaderUtils = require("loader-utils");

module.exports = function(content) {
  const url = loaderUtils.interpolateName(this, "[hash].[ext]", {
    content,
  });
  const { name } = this.getOptions();
  const url = loaderUtils.interpolateName(this, name, { content });

  ...
};
```

After running (node ./run-loader.js), you should see something:

```
{
  result: [ 'export default __webpack_public_path__ + "demo.txt";' ],
  resourceBuffer: <Buffer 66 6f 6f 62 61 72 0a>,
  cacheable: true,
  fileDependencies: [ './demo.txt' ],
  contextDependencies: [],
  missingDependencies: []
}
```

You can see that the result matches what the loader should have returned. You can try to pass more options to the loader or use query parameters to see what happens with different combinations.



It's a good idea to validate options and rather fail hard than silently if the options aren't what you expect. [schema-utils](https://www.npmjs.com/package/schema-utils)⁵ has been designed for this purpose.

⁵<https://www.npmjs.com/package/schema-utils>

33.6 Connecting custom loaders with webpack

To get the most out of loaders, you have to connect them with webpack. To achieve this, you can use imports:

src/component.js

```
import "!../loaders/demo-loader?name=foo!./main.css";
```

Given the definition is verbose, the loader can be aliased as below:

webpack.config.js

```
const commonConfig = merge([
  {
    resolveLoader: {
      alias: {
        "demo-loader": path.resolve(
          __dirname,
          "loaders/demo-loader.js"
        ),
      },
    },
  },
  ...
]);
```

With this change the import can be simplified:

```
import "!../loaders/demo-loader?name=foo!./main.css";
import "!demo-loader?name=foo!./main.css";
```

You could also handle the loader definition through `rules` and publish it as an npm package to consume.

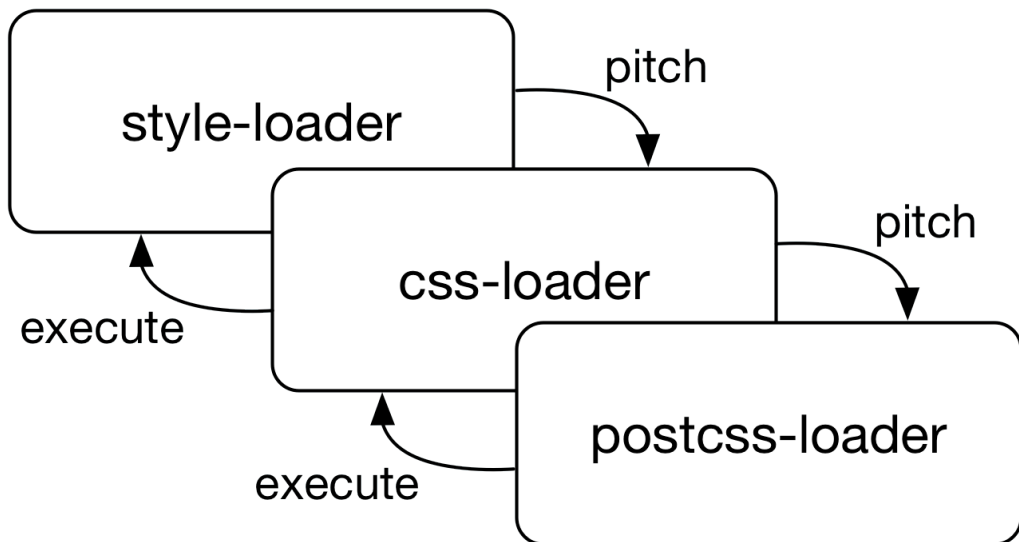


Although using **loader-runner** can be convenient for developing and testing loaders, implement integration tests that run against webpack. Subtle differences between environments make this essential. You can find a full testing setup at the *Extending with Plugins* chapter.



The [official documentation](https://webpack.js.org/api/loaders/)⁶ covers the loader API in detail. You can see all fields available through this there. For example, `mode` is exposed.

33.7 Pitch loaders



Webpack loader processing

Webpack evaluates loaders in two phases: pitching and evaluating. If you are used to web event semantics, these map to capturing and bubbling. The idea is that webpack allows you to intercept execution during the pitching (capturing) phase. It goes through the loaders left to right first and executes them from right to left after that.

⁶<https://webpack.js.org/api/loaders/>

A pitch loader allows you shape the request and even terminate it. Set it up:

loaders/pitch-loader.js

```
const loaderUtils = require("loader-utils");

module.exports = function (input) {
  return input + this.getOptions().text;
};

module.exports.pitch = function (remaining, preceding, input) {
  console.log(`Remaining: ${remaining}, preceding: ${preceding}`);
  console.log(`Input: ${JSON.stringify(input, null, 2)}`);
};

return "pitched";
};
```

To connect it to the runner, add it to the loader definition:

run-loader.js

```
runLoaders(
  {
    resource: "./demo.txt",
    loaders: [
      ...
      path.resolve(__dirname, "./loaders/pitch-loader"),
    ],
    ...
  },
  (err, result) => (err ? console.error(err) : console.log(result))
);
```

If you run (`node ./run-loader.js`) now, the pitch loader should log intermediate data and intercept the execution.

33.8 Caching with loaders

Although webpack caches loaders by default unless they set `this.cacheable(false)`, writing a caching loader can be a good exercise as it helps you to understand how loader stages can work together. The example below shows how to achieve this (courtesy of Vladimir Grenaderov):

```
const cache = new Map();

module.exports = function (content) {
  // Calls only once for given resourcePath
  const callbacks = cache.get(this.resourcePath);
  callbacks.forEach((callback) => callback(null, content));
  cache.set(this.resourcePath, content);

  return content;
};

module.exports.pitch = function () {
  if (cache.has(this.resourcePath)) {
    const item = cache.get(this.resourcePath);

    if (item instanceof Array) {
      item.push(this.async()); // Load to cache
    } else {
      return item; // Hit cache
    }
  } else {
    cache.set(this.resourcePath, []); // Missed cache
  }
};
```

A pitch loader can be used to attach metadata to the input to use later. In this example, a cache was constructed during the pitching stage, and it was accessed during normal execution.

33.9 Conclusion

Writing loaders is fun in the sense that they describe transformations from a format to another. Often you can figure out how to achieve something specific by either studying either the API documentation or the existing loaders.

To recap:

- **loader-runner** is a valuable tool for understanding how loaders work. Use it for debugging how loaders work.
- Webpack **loaders** accept input and produce output based on it.
- Loaders can be either synchronous or asynchronous. In the latter case, you should use `this.async()` webpack API to capture the callback exposed by webpack.
- If you want to generate code dynamically for webpack entries, that's where loaders can come in handy. A loader does not have to accept input. It's acceptable that it returns only output in this case.
- When developing loaders locally, consider setting up a `resolveLoader.alias` to clean up references.
- Pitching stage complements the default behavior allowing you to intercept and to attach metadata.

You'll learn to write plugins in the next chapter. Plugins allow you to intercept webpack's execution process, and they can be combined with loaders to develop more advanced functionality.

34. Extending with Plugins

Compared to loaders, plugins are a more flexible means to extend webpack. You have access to webpack's **compiler** and **compilation** processes. It's possible to run child compilers, and plugins can work in tandem with loaders as `MiniCssExtractPlugin` shows.

Plugins allow you to intercept webpack's execution through hooks. Webpack itself has been implemented as a collection of plugins. Underneath it relies on [tapable](#)¹ plugin interface that allows webpack to apply plugins in different ways.

You'll learn to develop a couple of small plugins next. Unlike for loaders, there is no separate environment where you can run plugins, so you have to run them against webpack itself. It's possible to push smaller logic outside of the webpack facing portion, though, as this allows you to unit test it in isolation.

34.1 The basic flow of webpack plugins

A webpack plugin is expected to expose an `apply(compiler)` method. JavaScript allows multiple ways to do this. You could use a function and then attach methods to its prototype. To follow the newest syntax, you could use a `class` to model the same idea.

Regardless of your approach, you should capture possible options passed by a user at the constructor. It's a good idea to declare a schema to communicate them to the user. [schema-utils](#)² allows validation and works with loaders too.

When the plugin is connected to webpack configuration, webpack will run its constructor and call `apply` with a compiler object passed to it. The object exposes webpack's plugin API and allows you to use its hooks as listed by [the official compiler reference](#)³.

¹<https://www.npmjs.com/package/tapable>

²<https://www.npmjs.com/package/schema-utils>

³<https://webpack.js.org/api/plugins/compiler/>

34.2 Setting up a development environment

To test and develop plugins against webpack, a good practice is to set up a harness that captures file output in-memory so you can assert output. You can also validate output against webpack stats.

The trick is to use [memfs](https://www.npmjs.com/package/memfs)⁴ in combination with `compiler.outputFileSystem`. Install **memfs** first:

```
npm add memfs -D
```

Implement a test bootstrap:

plugins/test.js

```
const webpack = require("webpack");
const { createFsFromVolume, Volume } = require("memfs");

// The compiler helper accepts filenames should be in the output
// so it's possible to assert the output easily.
function compile(config, filenames = []) {
  return new Promise((resolve, reject) => {
    const compiler = webpack(config);
    compiler.outputFileSystem = createFsFromVolume(new Volume());
    const memfs = compiler.outputFileSystem;

    compiler.run((err, stats) => {
      if (err) {
        return reject(err);
      }

      // Now only errors are captured from stats.
      // It's possible to capture more to assert.
      if (stats.hasErrors()) {
        return reject(stats.toString("errors-only"));
      }
    });
  });
}
```

⁴<https://www.npmjs.com/package/memfs>

```

    }

    const ret = {};
    filenames.forEach((filename) => {
      // The assumption is that webpack outputs behind ./dist.
      ret[filename] = memfs.readFileSync(`./dist/${filename}`, {
        encoding: "utf-8",
      });
    });
    return resolve(ret);
  });
}

async function test() {
  console.log(
    await compile({
      entry: "./test-entry.js",
    })
  );
}

test();

```

In addition, set up a test entry:

plugins/test-entry.js

```
console.log("hello from entry");
```



See [Stack Overflow](https://stackoverflow.com/questions/39923743/is-there-a-way-to-get-the-output-of-webpack-node-api-as-a-string)⁵ for related discussion.

⁵<https://stackoverflow.com/questions/39923743/is-there-a-way-to-get-the-output-of-webpack-node-api-as-a-string>

34.3 Implementing a basic plugin

The most basic plugin should do two things: capture options and provide apply method:

plugins/demo-plugin.js

```
module.exports = class DemoPlugin {  
  apply() {  
    console.log("applying");  
  }  
};
```

To test the plugin, connect it to our test environment:

plugins/test.js

```
...  
const DemoPlugin = require("./demo-plugin");  
  
...  
  
async function test() {  
  console.log(  
    await compile({  
      entry: "./test-entry.js",  
      plugins: [new DemoPlugin()],  
    })  
  );  
}
```

If you run the test (`node ./test.js`), you should see applying message at the console. Given most plugins accept options, it's a good idea to capture those and pass them to apply.

34.4 Capturing options

Options can be captured through a constructor:

plugins/demo-plugin.js

```
module.exports = class DemoPlugin {  
  constructor(options) {  
    this.options = options;  
  }  
  apply() {  
    console.log("apply", this.options);  
  }  
};
```

Running the plugin now would result in `apply undefined` kind of message given no options were passed.

Adjust the configuration to pass an option:

plugins/test.js

```
async function test() {  
  console.log(  
    await compile({  
      entry: "./test-entry.js",  
      plugins: [new DemoPlugin({ name: "demo" })],  
    })  
  );  
}
```

Now you should see `apply { name: 'demo' }` after running.

34.5 Understanding compiler and compilation

apply receives webpack's compiler as a parameter. Adjust as below:

plugins/demo-plugin.js

```
module.exports = class DemoPlugin {  
  constructor(options) {  
    this.options = options;  
  }  
  apply(compiler) {  
    console.log(compiler);  
  }  
};
```

After running, you should see a lot of data. Especially options should look familiar as it contains webpack configuration. You can also see familiar names like records.

If you go through webpack's [plugin development documentation](https://webpack.js.org/api/plugins/)⁶, you'll see a compiler provides a large number of hooks. Each hook corresponds to a specific stage. For example, to emit files, you could listen to the emit event and then write.

Change the implementation to listen and capture compilation:

plugins/demo-plugin.js

```
module.exports = class DemoPlugin {  
  constructor(options) {  
    this.options = options;  
  }  
  apply(compiler) {  
    compiler.hooks.thisCompilation.tap(  
      "DemoPlugin",  
      (compilation) => console.log(compilation)  
    );  
  }  
};
```

⁶<https://webpack.js.org/api/plugins/>

Running the build should show more information than before because a compilation object contains the whole dependency graph traversed by webpack. You have access to everything related to it here, including entries, chunks, modules, assets, and more.



Many of the available hooks expose compilation, but sometimes they reveal a more specific structure, and it takes a more particular study to understand those.

34.6 Writing files through compilation

The `assets` object of compilation can be used for writing new files. You can also capture already created assets, manipulate them, and write them back.

To write an asset, you have to use [webpack-sources](https://www.npmjs.com/package/webpack-sources)⁷ file abstraction. It's included to webpack by default starting from version 5.

Adjust the code as follows to write through `RawSource`:

plugins/demo-plugin.js

```
const { sources, Compilation } = require("webpack");

module.exports = class DemoPlugin {
  constructor(options) {
    this.options = options;
  }
  apply(compiler) {
    const pluginName = "DemoPlugin";
    const { name } = this.options;

    compiler.hooks.thisCompilation.tap(
      pluginName,
      (compilation) => {
        compilation.hooks.processAssets.tap(
```

⁷<https://www.npmjs.com/package/webpack-sources>

```
    {
      name: pluginName,
      // See lib/Compilation.js in webpack for more
      stage: Compilation.PROCESS_ASSETS_STAGE_ADDITIONAL,
    },
  ) =>
    compilation.emitAsset(
      name,
      new sources.RawSource("hello", true)
    )
  );
}
);
}
};
```

To make sure the file was emitted, adjust the test:

plugins/test.js

```
async function test() {
  console.log(
    await compile(
      {
        entry: "./test-entry.js",
        plugins: [new DemoPlugin({ name: "demo" })],
      },
      ["demo"]
    )
  );
}
```

If you run the test again (node ./test.js), you should see { demo: 'hello' } in the console output.



Compilation has a set of hooks of its own as covered in [the official compilation reference](#)⁸.

34.7 Managing warnings and errors

Plugin execution can be caused to fail by throwing (`throw new Error("Message")`). If you validate options, you can use this method.

In case you want to give the user a warning or an error message during compilation, you should use `compilation.warnings` and `compilation.errors`. Example:

```
compilation.warnings.push("warning");  
compilation.errors.push("error");
```

There's a logging API that lets you pass messages to webpack. Consider the API below:

```
const logger = compiler.getInfrastructureLogger("Demo Plugin");  
logger.log("hello from compiler");
```

You can use the API familiar from console so `warning`, `error`, and `group` amongst other methods will work. See [the logging documentation](#)⁹ for further details.

34.8 Plugins can have plugins

A plugin can provide hooks of its own. [html-webpack-plugin](#)¹⁰ is a good example of a plugin providing its own plugin interface.

⁸<https://webpack.js.org/api/plugins/compiler/>

⁹<https://webpack.js.org/api/logging/>

¹⁰<https://www.npmjs.com/package/html-webpack-plugin>

34.9 Conclusion

When you begin to design a plugin, spend time studying existing plugins that are close enough. Develop plugins piece-wise so that you validate one piece at a time. Studying webpack source can give more insight, given it's a collection of plugins itself.

To recap:

- **Plugins** can intercept webpack's execution and extend it making them more flexible than loaders.
- Plugins can be combined with loaders. `MiniCssExtractPlugin` works this way. The accompanying loader is used to mark assets to extract.
- Plugins have access to webpack's **compiler** and **compilation** processes. Both provide hooks for different stages of webpack's execution flow and allow you to manipulate it. Webpack itself works this way.
- Plugins can emit new assets and shape existing assets.
- Plugins can implement plugin systems of their own. `HtmlWebpackPlugin` is an example of such a plugin.

Conclusion

As this book has demonstrated, webpack is a versatile tool. To make it easier to recap the content and techniques, go through the checklists below.

General checklist

- **Source maps** allow you to debug your code in the browser during development. They can also give better quality stack traces during production usage if you capture the output. The *Source Maps* chapter delves into the topic.
- To keep your builds fast, consider optimizing. The *Performance* chapter discusses a variety of strategies you can use to achieve this.
- To keep your configuration maintainable, consider composing it. As webpack configuration is JavaScript code, it can be arranged in many ways. The *Composing Configuration* chapter discusses the topic.
- The way webpack consumes packages can be customized. The *Consuming Packages* chapter covers specific techniques related to this.
- Sometimes you have to extend webpack. The *Extending with Loaders* and *Extending with Plugins* chapters show how to achieve this. You can also work on top of webpack's configuration definition and implement an abstraction of your own for it to suit your purposes.

Development checklist

- To get most out of webpack during development, use **webpack-plugin-serve** (WPS) or **webpack-dev-server** (WDS). You can also find middlewares which

you can attach to your Node server during development. The *Development Server* chapter covers both in greater detail.

- Webpack implements **Hot Module Replacement** (HMR). It allows you to replace modules without forcing a browser refresh while your application is running. The *Hot Module Replacement* appendix covers the topic in detail.
- Consider using *Module Federation* when a project gains complexity and it's using multiple different technologies or it has multiple teams working on various functionalities. The approach takes microservices to frontend development and allows you to align your frontend with microbackends.

Production checklist

Styling

- Webpack inlines style definitions to JavaScript by default. To avoid this, separate CSS to a file of its own using `MiniCssExtractPlugin` or an equivalent solution. The *Separating CSS* chapter covers how to achieve this.
- To decrease the number of CSS rules to write, consider **autoprefixing** your rules. The *Autoprefixing* chapter shows how to do this.
- Unused CSS rules can be eliminated based on static analysis. The *Eliminating Unused CSS* chapter explains the basic idea of this technique.

Assets

- When loading images through webpack, optimize them, so the users have less to download. The *Loading Images* chapter shows how to do this.
- Load only the fonts you need based on the browsers you have to support. The *Loading Fonts* chapter discusses the topic.
- Minify your source files to make sure the browser to decrease the payload the client has to download. The *Minifying* chapter shows how to achieve this.

Caching

- To benefit from client caching, split a vendor bundle out of your application. This way the client has less to download in the ideal case. The *Bundle Splitting*

chapter discusses the topic. The *Adding Hashes to Filenames* chapter shows how to achieve cache invalidation on top of that.

- Use webpack's **code splitting** functionality to load code on demand. The technique is handy if you don't need all the code at once and instead can push it behind a logical trigger such as clicking a user interface element. The *Code Splitting* chapter covers the technique in detail. The *Dynamic Loading* chapter shows how to handle more advanced scenarios.
- Add hashes to filenames as covered in the *Adding Hashes to Filenames* chapter to benefit from caching and separate a runtime to improve the solution further as discussed in the *Separating a Runtime* chapter.

Optimization

- Use ES2015 module definition to leverage **tree shaking**. It allows webpack to eliminate unused code paths through static analysis. See the *Tree Shaking* chapter for the idea.
- Set application-specific environment variables to compile it production mode. You can implement feature flags this way. See the *Environment Variables* chapter to recap the technique.
- Analyze build statistics to learn what to improve. The *Build Analysis* chapter shows how to do this against multiple available tools.
- Push a part of the computation to web workers. The *Web Workers* chapter covers how to achieve this.

Output

- Clean up and attach information about the build to the result. The *Tidying Up* chapter shows how to do this.

Conclusion

Webpack allows you to use a lot of different techniques to splice up your build. It supports multiple output formats as discussed in the *Output* part of the book. Despite its name, it's not only for the web. That's where most people use it, but the tool does far more than that.

Appendices

As not everything that's worth discussing fits into the main content, you can find related material in brief appendices. These support the primary content and explain specific topics, such as *Hot Module Replacement*, in greater detail. You will also learn to troubleshoot webpack.

Comparison of Build Tools

Back in the day, it was enough to concatenate scripts together. Times have changed, though, and distributing your JavaScript code can be a complicated endeavor. This problem has escalated with the rise of single-page applications (SPAs) as they tend to rely on many big libraries. For this reason, many loading strategies exist. The basic idea is to defer loading instead of loading all at once.

The popularity of Node and [npm](#)¹¹, its package manager, provide more context. Before npm became popular, it was hard to consume dependencies. There was a period when people developed frontend specific package managers, but npm won in the end. Now dependency management is more comfortable than before, although there are still challenges to overcome.



[Tooling.Report](#)¹² provides a feature comparison of the most popular build tools.

Task runners

Historically speaking, there have been many build tools. *Make* is perhaps the best known, and it's still a viable option. Specialized *task runners*, such as Grunt and Gulp were created particularly with JavaScript developers in mind. Plugins available through npm made both task runners powerful and extendable. It's possible to use even npm scripts as a task runner. That's common, particularly with webpack.

Make

[Make](#)¹³ goes way back, as 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

¹¹<https://www.npmjs.com/>

¹²<https://bundlers.tooling.report/>

¹³https://en.wikipedia.org/wiki/Make_%28software%29

purposes. For instance, you could have different 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 C in any way. James Coglan discusses in detail [how to use Make with JavaScript](https://blog.jcoglan.com/2014/02/05/building-javascript-projects-with-make/)¹⁴. 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
```

¹⁴<https://blog.jcoglan.com/2014/02/05/building-javascript-projects-with-make/>

With Make, you model your tasks using Make-specific syntax and terminal commands making it possible to integrate with webpack.

npm scripts as a task runner

Even though npm CLI wasn't primarily designed to be used as a task runner, it works as such thanks to `package.json` `scripts` field. Consider the example below:

`package.json`

```
{
  "scripts": {
    "start": "wp --mode development",
    "build": "wp --mode production",
    "build:stats": "wp --mode production --json > stats.json"
  }
}
```

These scripts can be listed using `npm run` and then executed using `npm run <script>`. You can also namespace your scripts using a convention like `test:watch`. The problem with this approach is that it takes care to keep it cross-platform.

Instead of `rm -rf`, you likely want to use utilities such as [rimraf](https://www.npmjs.com/package/rimraf)¹⁵ and so on. It's possible to invoke other tasks runners here to hide the fact that you are using one. This way you can refactor your tooling while keeping the interface as the same.

Grunt

[Grunt](http://gruntjs.com/)¹⁶ was the first famous task runner for frontend developers. Its plugin architecture contributed towards its popularity. Plugins are often complicated by themselves. As a result, when configuration grows, it can become tricky to understand what's going on.

¹⁵<https://www.npmjs.com/package/rimraf>

¹⁶<http://gruntjs.com/>

Here's an example from [Grunt documentation](#)¹⁷. In this configuration, you define a linting and watcher tasks. When the *watch* task gets run, it triggers the *lint* task as well. This way, as you run Grunt, you get warnings in real-time in the terminal as you edit the source code.

Gruntfile.js

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

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

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

In practice, you would have many small tasks for specific purposes, such as building the project. An essential part of the power of Grunt is that it hides a lot of the wiring from you.

Taken too far, this can get problematic. It can become hard to understand what's going on under the hood. That's the architectural lesson to take from Grunt.

¹⁷<http://gruntjs.com/sample-gruntfile>



`grunt-webpack`¹⁸ plugin allows you to use webpack in a Grunt environment while you leave the heavy lifting to webpack.

Gulp

`Gulp`¹⁹ takes a different approach. Instead of relying on configuration per plugin, you deal with actual code. If you are familiar with Unix and piping, you'll like Gulp. You have *sources* to match files, *filters* to operate on these sources, and *sinks* to pipe the build results.

Here's an abbreviated sample *Gulpfile* adapted from the project's README to give you a better idea of the approach:

Gulpfile.js

```
const gulp = require("gulp");
const coffee = require("gulp-coffee");
const concat = require("gulp-concat");
const uglify = require("gulp-uglify");
const sourcemaps = require("gulp-sourcemaps");
const del = require("del");

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

// Not all tasks need to use streams.
// A gulpfile is another node program
// and you can use all packages available on npm.
gulp.task("clean", () => del(["build"]));
```

¹⁸<https://www.npmjs.com/package/grunt-webpack>

¹⁹<http://gulpjs.com/>


```
gulp.task("scripts", ["clean"], () =>
  // Minify and copy all JavaScript (except vendor scripts)
  // with source maps all the way down.
  gulp
    .src(paths.scripts)
    // Pipeline within pipeline
    .pipe(sourcemaps.init())
    .pipe(coffee())
    .pipe(uglify())
    .pipe(concat("all.min.js"))
    .pipe(sourcemaps.write())
    .pipe(gulp.dest("build/js"))
);
gulp.task("watch", () => 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 hack it if you run into troubles. You can wrap existing Node 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 newer approaches come in.



[webpack-stream](#)²⁰ allows you to use webpack in a Gulp environment.

Script loaders

For a while, [RequireJS](#)²¹, a script loader, was popular. The idea was to provide an asynchronous module definition and build on top of that. Fortunately, the standards have caught up, and RequireJS seems more like a curiosity now.

²⁰<https://www.npmjs.com/package/webpack-stream>

²¹<http://requirejs.org/>

RequireJS

RequireJS²² was perhaps the first script loader that became genuinely popular. It gave the first proper look at what modular JavaScript on the web could be. Its greatest attraction was AMD. It introduced a define wrapper:

```
define(["./MyModule.js"], function (MyModule) {  
    return function() {}; // Export at module root  
});  
  
// or  
define(["./MyModule.js"], function (MyModule) {  
    return {  
        hello: function() {...}, // Export as a module function  
    };  
});
```

Incidentally, it's possible to use `require` within the wrapper:

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

This latter approach eliminates a part of the clutter. You still end up with code that feels redundant. ES2015 and other standards solve this.



Jamund Ferguson has written an excellent blog series on how to port from RequireJS to webpack²³.

²²<http://requirejs.org/>

²³<https://gist.github.com/xjamundx/b1c800e9282e16a6a18e>

JSPM

Using [JSPM](http://jspm.io/)²⁴ is entirely different than previous tools. It comes with a command-line 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](https://github.com/systemjs/systemjs#plugins)²⁵ that allow you to load various formats to your project.

Bundlers

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. *bundlers*, such as Browserify, Brunch, or webpack, exist for this reason and they operate on a lower level of abstraction. Instead of operating on files, they operate on modules and assets.

Browserify

Dealing with JavaScript modules has always been a bit of a problem. The language itself didn't have the concept of modules till ES2015. Ergo, the language was stuck in the '90s when it comes to browser environments. Various solutions, including [AMD](http://requirejs.org/docs/whyamd.html)²⁶, have been proposed.

[Browserify](http://browserify.org/)²⁷ is one solution to the module problem. It allows CommonJS modules to be bundled together. You can hook it up with Gulp, and you can find smaller transformation tools that allow you to move beyond the basic usage. For example, [watchify](https://www.npmjs.com/package/watchify)²⁸ provides a file watcher that creates bundles for you during development saving effort.

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

²⁴<http://jspm.io/>

²⁵<https://github.com/systemjs/systemjs#plugins>

²⁶<http://requirejs.org/docs/whyamd.html>

²⁷<http://browserify.org/>

²⁸<https://www.npmjs.com/package/watchify>



[Splittable](https://www.npmjs.com/package/splittable)²⁹ is a Browserify wrapper that allows code splitting, supports ES2015 out of the box, tree shaking, and more. [bankai](https://www.npmjs.com/package/bankai)³⁰ is another option to consider.

Brunch

Compared to Gulp, [Brunch](http://brunch.io/)³¹ 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:

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

²⁹<https://www.npmjs.com/package/splittable>

³⁰<https://www.npmjs.com/package/bankai>

³¹<http://brunch.io/>

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

Rollup

[Rollup](https://www.npmjs.com/package/rollup)³² focuses on bundling ES2015 code. *Tree shaking* is one of its selling points and it supports code splitting as well. You can use Rollup with webpack through [rollup-loader](https://www.npmjs.com/package/rollup-loader)³³.

[vite](https://www.npmjs.com/package/vite)³⁴ is an opinionated wrapper built on top of Rollup and it has been designed especially with Vue 3 in mind. [nollup](https://www.npmjs.com/package/nollup)³⁵ is another wrapper and it comes with features like *Hot Module Replacement* out of the box.

Webpack

You could say [webpack](https://www.npmjs.com/package/webpack)³⁶ takes a more unified 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.

Webpack core can be extended using specific *loaders* and *plugins*. It gives control over how it *resolves* the modules, making it possible to adapt your build to match specific situations and workaround packages that don't work correctly out of the box.

Compared to the other tools, webpack comes with initial complexity, but it makes up for this through its broad feature set. It's an advanced tool that requires patience. But once you understand the basic ideas behind it, webpack becomes powerful.

To make it easier to use, tools such as [create-react-app](https://www.npmjs.com/package/create-react-app)³⁷, [poi](https://poi.js.org/)³⁸, and [instapack](https://www.npmjs.com/package/instapack)³⁹ have been built around it.

³²<https://www.npmjs.com/package/rollup>

³³<https://www.npmjs.com/package/rollup-loader>

³⁴<https://www.npmjs.com/package/vite>

³⁵<https://www.npmjs.com/package/nollup>

³⁶<https://webpack.js.org/>

³⁷<https://www.npmjs.com/package/create-react-app>

³⁸<https://poi.js.org/>

³⁹<https://www.npmjs.com/package/instapack>

Vite

[Vite](https://vitejs.dev/)⁴⁰ is tool comparable to webpack. It comes with features like lazy loading, ESM, JSX, and TypeScript support out of the box. The build functionality relies on Rollup and the development server is custom code. Originally it was developed with Vue in mind but since the scope of the tool has grown to support popular frameworks like React. It's possible to extend the tool using Vite specific plugins and also Rollup plugins are supported making it a versatile solution.

Zero configuration bundlers

There's a whole category of *zero configuration* bundlers. The idea is that they work out of the box without any extra setup. [Parcel](https://parceljs.org/)⁴¹ is perhaps the famous of them.

[FuseBox](https://www.npmjs.com/package/fuse-box)⁴² is a bundler focusing on speed. It uses a zero-configuration approach and aims to be usable out of the box.

These tools include [microbundle](https://www.npmjs.com/package/microbundle)⁴³, [bili](https://www.npmjs.com/package/bili)⁴⁴, [asbundle](https://www.npmjs.com/package/asbundle)⁴⁵, and [tsdx](https://www.npmjs.com/package/tsdx)⁴⁶.

Other Options

You can find more alternatives as listed below:

- [Rome](https://romefrontend.dev/)⁴⁷ is an entire toolchain built around the problems of linting, compiling, and bundling.
- [esbuild](https://www.npmjs.com/package/esbuild)⁴⁸ is a performance-oriented bundler written in Go.

⁴⁰<https://vitejs.dev/>

⁴¹<https://parceljs.org/>

⁴²<https://www.npmjs.com/package/fuse-box>

⁴³<https://www.npmjs.com/package/microbundle>

⁴⁴<https://www.npmjs.com/package/bili>

⁴⁵<https://www.npmjs.com/package/asbundle>

⁴⁶<https://www.npmjs.com/package/tsdx>

⁴⁷<https://romefrontend.dev/>

⁴⁸<https://www.npmjs.com/package/esbuild>

- [AssetGraph](#)⁴⁹ takes an entirely different approach and builds on top of HTML semantics making it ideal for [hyperlink analysis](#)⁵⁰ or [structural analysis](#)⁵¹. [webpack-assetgraph-plugin](#)⁵² bridges webpack and AssetGraph together.
- [StealJS](#)⁵³ is a dependency loader and a build tool focusing on performance and ease of use.
- [Blendid](#)⁵⁴ is a blend of Gulp and bundlers to form an asset pipeline.
- [swc](#)⁵⁵ is a JavaScript/TypeScript compiler focusing on performance written in Rust.
- [Packem](#)⁵⁶ is another Rust based option for bundling JavaScript.
- [Sucrase](#)⁵⁷ is a light JavaScript/TypeScript compiler focusing on performance and recent language features.

Conclusion

Historically there have been a lot of build tools for JavaScript. Each has tried to solve a specific problem in its way. The standards have begun to catch up, and less effort is required around basic semantics. Instead, tools can compete on a higher level and push towards better user experience. Often you can use a couple of separate solutions together.

To recap:

- **Task runners** and **bundlers** solve different problems. You can achieve similar results with both, but often it's best to use them together to complement each other.
- Older tools, such as Make or RequireJS, still have influence even if they aren't as popular in web development as they once were.
- Bundlers like Browserify or webpack solve an important problem and help you to manage complex web applications.

⁴⁹<https://www.npmjs.com/package/assetgraph>

⁵⁰<https://www.npmjs.com/package/hyperlink>

⁵¹<https://www.npmjs.com/package/assetviz>

⁵²<https://www.npmjs.com/package/webpack-assetgraph-plugin>

⁵³<https://stealjs.com/>

⁵⁴<https://www.npmjs.com/package/blendid>

⁵⁵<https://swc-project.github.io/>

⁵⁶<https://packem.github.io/>

⁵⁷<https://www.npmjs.com/package/sucrase>

- Emerging technologies approach the problem from different angles. Sometimes they build on top of other tools, and at times they can be used together.

Hot Module Replacement

Hot Module Replacement (HMR) builds on top of the WDS. It enables an interface that makes it possible to swap modules live. For example, **style-loader** can update your CSS without forcing a refresh. Implementing HMR for styles is ideal because CSS is stateless by design.

HMR is possible with JavaScript too, but due to application state, it's harder. [react-refresh-webpack-plugin](#)⁵⁸ and [vue-hot-reload-api](#)⁵⁹ are good examples.



Given HMR can be complex to implement, a good compromise is to store application state to `localStorage` and then hydrate the application based on that after a refresh. Doing this pushes the problem to the application side.

Enabling HMR

The following steps need to be enabled for HMR to work:

1. The development server has to run in the hot mode to expose the hot module replacement interface to the client.
2. Webpack has to provide hot updates to the server and can be achieved using `webpack.HotModuleReplacementPlugin`.
3. The client has to run specific scripts provided by the development server. They will be injected automatically but can be enabled explicitly through entry configuration.
4. The client has to implement the HMR interface through `module.hot.accept` and optionally `module.hot.dispose` to clean module before replacing it.

⁵⁸<https://www.npmjs.com/package/react-refresh-webpack-plugin>

⁵⁹<https://www.npmjs.com/package/vue-hot-reload-api>

Using `webpack-dev-server --hot` or running **webpack-plugin-serve** in hot mode solves the first two problems. In this case, you have to handle only the last one yourself if you want to patch JavaScript application code. Skipping the `--hot` flag and going through webpack configuration gives more flexibility.

The following listing contains the essential parts related to this approach. You will have to adapt from here to match your configuration style:

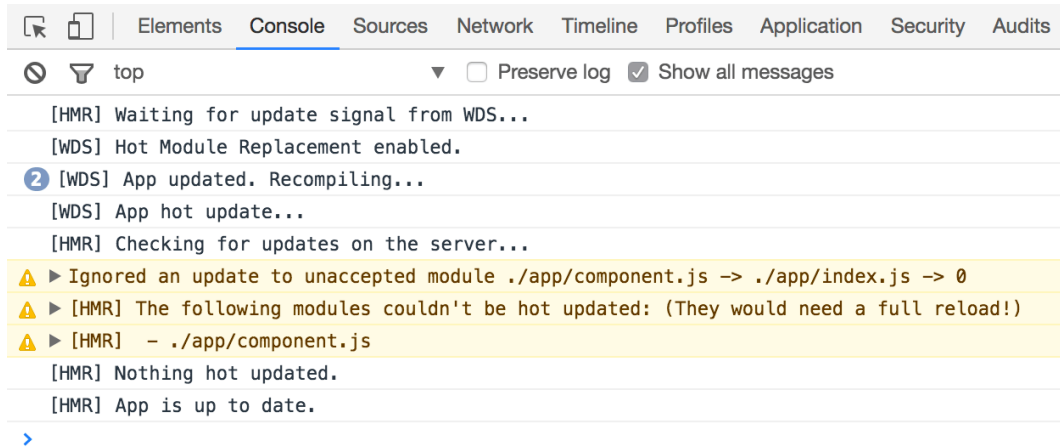
```
{
  devServer: {
    // Don't refresh if hot loading fails. Good while
    // implementing the client interface.
    hotOnly: true,

    // If you want to refresh on errors too, set
    // hot: true,
  },
  plugins: [
    // Enable the plugin to let webpack communicate changes
    // to WDS. --hot sets this automatically!
    new webpack.HotModuleReplacementPlugin(),
  ],
}
```



Starting from webpack 5, there's an alternative to `module.hot.import.meta.webpackHot` has been designed with ES2015 modules and Node mjs file extension in mind as it doesn't allow mixing CommonJS and ES2015 syntax.

If you implement configuration like above without implementing the client interface, you will most likely end up with an error:



```
[HMR] Waiting for update signal from WDS...
[WDS] Hot Module Replacement enabled.
2 [WDS] App updated. Recompiling...
[WDS] App hot update...
[HMR] Checking for updates on the server...
⚠ ▶ Ignored an update to unaccepted module ./app/component.js -> ./app/index.js -> 0
⚠ ▶ [HMR] The following modules couldn't be hot updated: (They would need a full reload!)
⚠ ▶ [HMR] - ./app/component.js
[HMR] Nothing hot updated.
[HMR] App is up to date.
```

No refresh

The message tells that even though the HMR interface notified the client portion of the code of a hot update, nothing was done about it and this is something to fix next.



The setup assumes you have set `optimization.moduleIds = 'named'`. If you run webpack in development mode, it will be on by default.



You should **not** enable HMR for your production configuration. It likely works, but it makes your bundles bigger than they should be.



If you are using Babel, configure it so that it lets webpack control module generation as otherwise, HMR logic won't work! See the *Loading JavaScript* chapter for the exact setup.

Implementing the HMR interface

Webpack exposes the HMR interface through a global variable: `module.hot`. It provides updates through `module.hot.accept(<path to watch>, <handler>)` function and you need to patch the application there.

The following implementation illustrates the idea against the tutorial application:

src/index.js

```
import component from "../component";

let demoComponent = component();

document.body.appendChild(demoComponent);

// HMR interface
if (module.hot) {
  // Capture hot update
  module.hot.accept("../component", () => {
    const nextComponent = component();

    // Replace old content with the hot loaded one
    document.body.replaceChild(nextComponent, demoComponent);

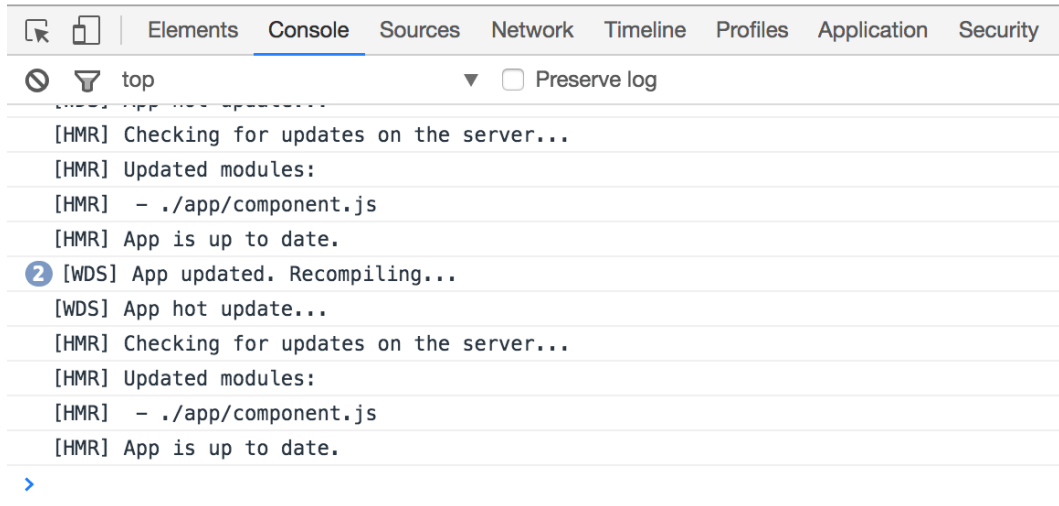
    demoComponent = nextComponent;
  });
}
```

If you refresh the browser, try to modify `src/component.js` after this change, and alter the text to something else, you should notice that the browser does not refresh at all. Instead, it should replace the DOM node while retaining the rest of the application as is.



`module.hot.accept` works with an array of filenames as well. The handler (second parameter) is optional.

The image below shows possible output:



```
[HMR] Checking for updates on the server...
[HMR] Updated modules:
[HMR]   - ./app/component.js
[HMR] App is up to date.
2 [WDS] App updated. Recompiling...
[WDS] App hot update...
[HMR] Checking for updates on the server...
[HMR] Updated modules:
[HMR]   - ./app/component.js
[HMR] App is up to date.
```

Patched a module successfully through HMR

The idea is the same with styling, React, Redux, and other technologies. Sometimes you don't have to implement the interface yourself even as available tooling takes care of that for you.



To prove that HMR retains application state, set up a [checkbox](#)⁶⁰ based component next to the original. The `module.hot.accept` code has to evolve to capture changes to it as well.



The `if(module.hot)` block is eliminated entirely from the production build as minifier picks it up. The *Minifying* chapter delves deeper into this topic.

⁶⁰<https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input/checkbox>



[hot-accept-webpack-plugin](#)⁶¹ and [module-hot-accept-loader](#)⁶² allow you to write `if (module.hot) { module.hot.accept(); }` for each module that was matched. It's useful in case you have modules that should accept hot loading without implementing the patching behavior.



[Deep dive into Hot Module Replacement by Stanimira Vlaeva](#)⁶³ discusses the topic in greater detail.

Setting WDS entry points manually

In the setup above, the WDS-related entries were injected automatically. Assuming you are using WDS through Node, you would have to set them yourself as the Node API doesn't support injecting. The example below illustrates how to achieve this:

```
entry: {
  hmr: [
    // Include the client code. Note host/post.
    "webpack-dev-server/client?http://localhost:8080",

    // Hot reload only when compiled successfully
    "webpack/hot/only-dev-server",

    // Alternative with refresh on failure
    // "webpack/hot/dev-server",
  ],
  ...
},
```

⁶¹<https://www.npmjs.com/package/hot-accept-webpack-plugin>

⁶²<https://www.npmjs.com/package/module-hot-accept-loader>

⁶³<https://nativescript.org/blog/deep-dive-into-hot-module-replacement-with-webpack-part-two-handling-updates/>

HMR and dynamic loading

Dynamic Loading through `require.context` and HMR requires extra effort:

```
const req = require.context("./pages", true, /^(.*\.(jsx$))[^.]*$/g);  
  
module.hot.accept(req.id, ...); // Replace modules here as above
```

Conclusion

HMR is one of those aspects of webpack that makes it attractive for developers and webpack has taken its implementation far. To work, HMR requires both client and server-side support. For this purpose, webpack-dev-server provides both. You will have to take care with the client-side, though, and either find a solution that implements the HMR interface or implement it yourself.

CSS Modules

Perhaps the most significant challenge of CSS is that all rules exist within **global scope**, meaning that two classes with the same name will collide. The limitation is inherent to the CSS specification, but projects have workarounds for the issue. [CSS Modules](#)⁶⁴ introduces **local scope** for every module by making every class declared within unique by including a hash in their name that is globally unique to the module.

CSS Modules through css-loader

Webpack's **css-loader** supports CSS Modules. You can enable it through a loader definition as above while enabling the support:

```
{
  use: {
    loader: "css-loader",
    options: {
      modules: true,
    },
  },
},
```

After this change, your class definitions remain local to the files. In case you want global class definitions, you need to wrap them within `:global(.redButton) { ... }` kind of declarations.

⁶⁴<https://github.com/css-modules/css-modules>

In this case, the `import` statement gives you the local classes you can then bind to elements. Assume you had CSS as below:

app/main.css

```
body {  
  background: cornsilk;  
}  
  
.redButton {  
  background: red;  
}
```

You could then bind the resulting class to a component:

app/component.js

```
import styles from './main.css';  
  
...  
  
// Attach the generated class name  
element.className = styles.redButton;
```

`body` remains as a global declaration still. It's that `redButton` that makes the difference. You can build component-specific styles that don't leak elsewhere this way.

CSS Modules allows composition to make it easier to work with your styles and you can also combine it with other loaders as long as you apply them before **css-loader**.



CSS Modules behavior can be modified [as discussed in the official documentation](#)⁶⁵. You have control over the names it generates for instance.



[eslint-plugin-css-modules](#)⁶⁶ is handy for tracking CSS Modules related problems.

⁶⁵<https://www.npmjs.com/package/css-loader#modules>

⁶⁶<https://www.npmjs.com/package/eslint-plugin-css-modules>

Using CSS Modules with third-party libraries and CSS

If you are using CSS Modules in your project, you should process standard CSS through a separate loader definition without the `modules` option of `css-loader` enabled. Otherwise, all classes will be scoped to their module. In the case of third-party libraries, this is almost certainly not what you want.

You can solve the problem by processing third-party CSS differently through an `include` definition against `node_modules`. Alternately, you could use a file extension (`.mcss`) to tell files using CSS Modules apart from the rest and then manage this situation in a loader test.

Conclusion

CSS Modules solve the scoping problem of CSS by defaulting to local scope per file. You can still have global styling, but it requires additional effort. Webpack can be set up to support CSS Modules easily as seen above.

Searching with React

Let's say you want to implement a rough little search for an application without a proper backend. You could do it through [lunr](http://lunrjs.com/)⁶⁷ and generate a static search index to serve.

The problem is that the index can be sizable depending on the amount of the content. The good thing is that you don't need the search index straight from the start. You can do something smarter instead. You can start loading the index when the user selects a search field.

Doing this defers the loading and moves it to a place where it's more acceptable for performance. The initial search is going to be slower than the subsequent ones, and you should display a loading indicator. But that's fine from the user's point of view. Webpack's *Code Splitting* feature allows doing this.

Implementing search with code splitting

To implement code splitting, you need to decide where to put the split point, put it there, and then handle the Promise:

```
import("./asset").then(asset => ...).catch(err => ...)
```

The beautiful thing is that this gives error handling in case something goes wrong (network is down, etc.) and gives a chance to recover. You can also use Promise based utilities like `Promise.all` for composing more complicated queries.

⁶⁷<http://lunrjs.com/>

In this case, you need to detect when the user selects the search element, load the data unless it has been loaded already, and then execute search logic against it. Consider the React implementation below:

App.js

```
import React from "react";

const App = () => {
  const [index, setIndex] = React.useState(null);
  const [value, setValue] = React.useState("");
  const [lines, setLines] = React.useState([]);
  const [results, setResults] = React.useState([]);

  const search = (lines, index, query) =>
    index.search(query.trim()).map((match) => lines[match.ref]);

  const onChange = ({ target: { value } }) => {
    setValue(value);

    // Search against lines and index if they exist
    if (lines && index) {
      setResults(search(lines, index, value));

      return;
    }

    // If the index doesn't exist, it has to be set it up.
    // You could show loading indicator here as loading might
    // take a while depending on the size of the index.
    loadIndex()
      .then(({ index, lines }) => {
        setIndex(index);
        setLines(lines);
        setResults(search(lines, index, value));
      })
      .catch((err) => console.error(err));
  }
}
```

```

    };

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

  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 `import` to tell Webpack
    // to split here and load our search index dynamically.
    //
    // Shim Promise.all for older browsers and Internet Explorer!
    return Promise.all([
      import("lunr"),
    ]
  )
}

```

```
import("../search_index.json"),
]).then(([ { Index }, { index, lines } ]) => ({
  index: Index.load(index),
  lines,
})));
}
```

In the example, webpack detects the `import` statically. It can 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.

Conclusion

Beyond search, the approach can be used with routers too. As the user enters a route, you can load the dependencies the resulting view needs. Alternately, you can start loading dependencies as the user scrolls a page and gets adjacent parts with actual functionality. `import` provides a lot of power and allows you to keep your application lean.

You can find [the full example](#)⁶⁸ showing how it all goes together with `lunr`, `React`, and `webpack`. The basic idea is the same, but there's more setup in place.

To recap:

- If your dataset is small and static, client-side search is a good option.
- You can index your content using a solution like `lunr`⁶⁹ and then perform a search against it.
- Webpack's **code splitting** feature is ideal for loading a search index on demand.
- Code splitting can be combined with a UI solution like `React` to implement the whole user interface.

⁶⁸<https://github.com/survivejs-demos/lunr-demo>

⁶⁹<http://lunrjs.com/>

Troubleshooting

Using webpack can lead to a variety of runtime warnings or errors. Often a particular part of the build fails for a reason or another. A basic process can be used to figure out these problems:

1. Enable `stats.errorDetails` in webpack configuration to get more information.
2. Study the origin of the error carefully. Sometimes you can infer what's wrong with context. If webpack fails to parse a module, it's likely not passing it through a loader you expect for example.
3. Try to understand where the error stems. Does it come from your code, a dependency, or webpack?
4. Remove code until the error goes away and add code back till it appears again. Simplify as much as possible to isolate the problem.
5. If the code worked in another project, figure out what's different. It's possible the dependencies between the projects vary, or the setup differs somehow. At the worst case, a package you rely upon has gained a regression. Using a `lockfile` is a good idea for this reason.
6. Study the related packages carefully. Looking into the package `package.json` can yield insight. It's possible the package you are using does not resolve the way you expect.
7. Search for the error online. Perhaps someone else has run into it. [Stack Overflow](#)⁷⁰ and [the official issue tracker](#)⁷¹ are good starting points.
8. Enable `stats: "verbose"` to get more information out of webpack. The [official documentation covers more flags](#)⁷².
9. Add a temporary `console.log` near the error to get more insight into the problem. A heavier option is to [debug webpack through Chrome Dev Tools](#)⁷³.
10. [Ask a question at Stack Overflow](#)⁷⁴ or [use the official Gitter channel](#)⁷⁵.

⁷⁰<https://stackoverflow.com/questions/tagged/webpack>

⁷¹<https://github.com/webpack/webpack/issues>

⁷²<https://webpack.js.org/configuration/stats/>

⁷³<https://medium.com/webpack/webpack-bits-learn-and-debug-webpack-with-chrome-dev-tools-da1c5b19554>

⁷⁴<https://stackoverflow.com/questions/tagged/webpack>

⁷⁵<https://gitter.im/webpack/webpack>

11. If everything fails and you are convinced you have found a bug, report the problem at the issue tracker that's closest to it. Follow the issue template carefully, and provide a minimal runnable example as that will help the maintainers.

Sometimes it's fastest to drop the error to a search engine and gain an answer that way. Other than that this is an excellent debugging order. If your setup worked in the past, you could also consider using commands like `git bisect`⁷⁶ to figure out what has changed between the known working state and the current broken one.

You'll learn about the most common errors next and how to deal with them.

Module related errors

Webpack emits various module related errors. I've listed the main ones and how to resolve them here.

Entry module not found

You can get `ERROR in Entry module not found` if you make an entry path point at a place that does not exist. The error message tells you what path webpack fails to find.

Module not found

You can receive `ERROR ... Module not found` in two ways. Either by breaking a loader definition so that it points to a loader that does not exist or by breaking an import path within your code so that it leads to a module that doesn't exist. The message points out what to fix.

Module parse failed

Even though webpack could resolve to your modules fine, it can still fail to build them and that's when you likely receive a `Module parse failed` error. This case can happen if you are using syntax that your loaders don't understand. You could be missing something in your processing pass.

⁷⁶<https://git-scm.com/docs/git-bisect>

Loader not found

There's another subtle loader related error, `Loader Not Found`. If a package matching to a loader name that does not implement the loader interface exists, webpack matches to that and gives a runtime error that says the package is not a loader.

If you write `loader: "eslint"` instead of `loader: "eslint-loader"`, you'll receive this error. If the package doesn't exist at all, then `Module not found` error will be raised.

Module build failed: Unknown word

`Module build failed: Unknown word` fits the same category. Parsing the file succeeded, but there was the unknown syntax. Most likely the problem is a typo, but this error can also occur when Webpack has followed an import and encountered syntax it doesn't understand. Most likely this means that a loader is missing for that particular file type.

SyntaxError: Unexpected token

`SyntaxError` is another error for the same category. This error is possible if you use ES2015 syntax that hasn't been transpiled alongside terser. As it encounters a syntax construct it does not recognize, it raises an error.

DeprecationWarning

Node may give a `DeprecationWarning` especially after webpack has been updated to a new major version. A plugin or a loader you are using may require updates. Often the changes required are minimal. To figure out where the warning is coming from, run webpack through Node: `node --trace-deprecation node_modules/.bin/wp --mode production`.

It's important to pass the `--trace-deprecation` flag to Node to see where the warning originates from. Using `--trace-warnings` is another way and it will capture the tracing information for all warnings, not only deprecations.

Conclusion

These are only examples of errors. Specific errors happen on the webpack side, but the rest comes from the packages it uses through loaders and plugins. Simplifying your project is a good step as that makes it easier to understand where the error happens.

In most cases, the errors are fast to solve if you know where to look, but in the worst case, you have come upon a bug to fix in the tooling. In that case, you should provide a high-quality report to the project and help to resolve it.

Glossary

Given webpack comes with specific terminology, the principal terms and their explanations have been gathered below.

- **Asset** is a general term for the media and source files of a project that are the raw material used by webpack in building a bundle.
- **Bundle** is the result of bundling. Bundling involves processing the source material of the application into a final bundle that is ready to use. A bundler can generate more than one bundle.
- **Bundle splitting** offers one way of optimizing a build, allowing webpack to generate multiple bundles for a single application. As a result, each bundle can be isolated from changes affecting others, reducing the amount of code that needs to be republished and therefore re-downloaded by the client and taking advantage of browser caching.
- **Chunk** is a webpack-specific term that is used internally to manage the bundling process. Webpack composes bundles out of chunks, and there are several types of those.
- **Code splitting** produces more granular bundles than bundle splitting. To use it, the developer has to enable it through specific calls in the source code. Using a `dynamic import()` is one way.
- **Entry** refers to a file used by webpack as a starting point for bundling. An application can have multiple entries and depending on configuration, each entry can result in multiple bundles. Entries are defined in webpack's entry configuration. Entries are **modules** at the beginning of the dependency graph.
- **Hashing** refers to the process of generating a hash that is attached to the asset/bundle path to invalidate it on the client. Example of a hashed bundle name: `app.f6f78b2fd2c38e8200d.js`.
- **Hot Module Replacement (HMR)** refers to a technique where code running in the browser is patched on the fly without requiring a full page refresh. When an application contains complex state, restoring it can be difficult without HMR or a similar solution.

- **Inlining** is the process of combining an asset within another. A good example is writing an image within a JavaScript file after encoding it to avoid emitting a separate file. Doing this avoids a roundtrip to the server and it can be a beneficial performance optimization in HTTP/1 environment.
- **Linting** relates to the process in which code is statically analyzed for a series of user-defined issues. These issues can range from discovering syntax errors to enforcing code-style. While linting is by definition limited in its capabilities, a linter is invaluable for helping with early error discovery and enforcing code consistency.
- **Loader** performs a transformation that accepts a source and returns transformed source. It can also skip processing and perform a check against the input instead. Through configuration, a loader targets a subset of modules, often based on the module type or location. A loader only acts on a single module at a time whereas a plugin can act on multiple files.
- **Minifying**, or minification, is an optimization technique in which code is written in a more compact form without losing meaning. Specific destructive transformations break code if you are not careful.
- **Module** is a general term to describe a piece of the application. In webpack, it can refer to JavaScript, a style sheet, an image or something else. **Loaders** allows webpack to support different file types and therefore different types of module. If you point to the same module from multiple places of a code base, webpack will generate a single module in the output which enables the singleton pattern on module level.
- **Module federation** is a technique that enables webpack to combine micro frontends developed separately as a single build.
- **Output** refers to files emitted by webpack. More specifically, webpack emits **bundles** and **assets** based on the output settings.
- **Plugins** connect to webpack's event system and can inject functionality into it. They allow webpack to be extended and can be combined with loaders for maximum control. Whereas a loader works on a single file, a plugin has much broader access and is capable of more global control.
- **Resolving** is the process that happens when webpack encounters a module or a loader. When that happens, it tries to resolve it based on the given resolution rules.
- **Source maps** describe the mapping between the source code and the generated code, allowing browsers to provide a better debugging experience. For exam-

ple, running ES2015 code through Babel generates completely new ES5 code. Without a source map, a developer would lose the link from where something happens in the generated code and where it happens in the source code. The same is true for style sheets when they run through a pre or post-processor.

- **Static analysis** - When a tool performs static analysis, it examines the code without running it which is how tools like ESLint or webpack operate. Statically analyzable standards, like ES2015 module definition, enable features like **tree shaking**.
- **Target** options of webpack allow you to override the default web target. You can use webpack to develop code for specific JavaScript platforms.
- **Tree shaking** is the process of dropping unused code based on static analysis. ES2015 module definition allows this process as it's possible to analyze in this particular manner.