Webpack 5 Up and Running

A quick and practical introduction to the JavaScript application bundler

Packt
www.packt.com

Tom Owens

Webpack 5 Up and Running

A quick and practical introduction to the JavaScript application bundler

Tom Owens

Packt

BIRMINGHAM - MUMBAI

Webpack 5 Up and Running

Commissioning Editor: Pavan Ramchandani
Acquisition Editor: Reshma Raman
Content Development Editor: Akhil Nair
Senior Editor: Hayden Edwards
Technical Editor: Deepesh Patel
Copy Editor: Safis Editing
Project Coordinator: Kinjal Bari
Proofreader: Safis Editing
Indexer: Priyanka Dhadke
Production Designer: Aparna Bhagat

First published: May 2020

Production reference: 1270520

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78995-440-1

`www.packt.com`

For my late father, who had a deep interest in my work as well as high technology,
allowing us to talk for hours on the subject.

Contributors

About the author

Tom Owens is a seasoned web developer with over 11 years of industry experience. He graduated from Liverpool John Moores University with a BA (Hons) and an FdA. Later, he would graduate with an MSc from the same institution. His career primarily began as a freelance web designer, which granted him industry experience around numerous related new media fields, including game development and, later, frontend development. He would eventually move into internet entrepreneurship, having garnered a great deal of knowledge in offering consultancy to highly successful individual entrepreneurs. After this, he was offered a role as Lecturer in Web Development for a private institution called SAE, who offered degrees to BSc (Hons) and BA (Hons) students, in partnership with Middlesex University London in the United Kingdom. This would help him cement himself as an expert in his field. He continues to have a passion for coaching and mentoring to this day. He is an active tutor, understanding the benefits of tutoring over lecturing, as it allows the structure to be more fluid, adapting to the bespoke needs of each student. He intends to bring that skill set to the fore in this book.

About the reviewers

Sonny Recio is an experienced developer with a seven-year track record of commended performance in software engineering. He recently built his team and solved technical problems on web and mobile apps for his clients.

Nowadays, Sonny spends more time learning and doing frontend development and creates rich user experiences in his projects, aside from managing his team and getting the hang of the business side of things.

He loves using React and is a big fan of functional programming. He reads books related to philosophy, biology, health, art, design, technology, and business in his free time.

Rajat Kumar is an engineer at Netflix, Inc. and has worked with many start-ups and Fortune 100 companies in the last 14 years. He regularly contributes to several open source projects and maintains Restify, an open source framework, built on Node.js.

Rajat enjoys writing code in JavaScript because he believes JavaScript has complexities and imperfections that mimic the real world. Currently, he is driving the architecture behind Netflix's dynamic asset bundling system built on top of webpack.

Aaron Czichon is Head of Software Development at Conclurer GmbH, as well as a conference speaker, author, and trainer with a focus on Node.js and web components.

Table of Contents

Preface

When I was asked to write this training book, I realized that very little was known about webpack and its purpose. It is generally something a developer stumbles upon and learns on the job, which can be a very laborious process. There is some documentation on the Webpack.js website as well as a handful of credible resources such as Medium. However, these resources do tend to speak to the reader from the point of view of an expert and I, for one, found this not to be ideal.

Coming from the background of being a lecturer in web development, I have seen how very skilled and intelligent people can have blind spots and knowledge gaps. As a lecturer, I was advised, and also relay the message on, that *there are no wrong questions*. Many people who have not come from a teaching background might suggest they don't want to speak down to you and would prefer you not ask the wrong questions. We have found this is harmful if a student would rather remain silent over asking questions.

I intend to keep things as simple as possible. I may have failed already using words such as "bespoke", nonetheless, the premise is that all of us have head-slapping moments, where we should have done something, and later on realize what we did wrong. Am I right? Well, it happens to the most brilliant of us. Also, most instructors may be reluctant to talk you through something to the nth degree for fear of patronizing you. The problem being is that there will always be some mundane detail, that the developer thought obvious but could be interpreted more than one way. When I lectured the rule was: "there are no stupid questions", so I hope to prove the necessity of that theory.

Who this book is for

The book is for web developers who wish to get started with dependency management in their web project by learning Webpack. Working knowledge of JavaScript is assumed.

What this book covers

`Chapter 1`, *Introduction to Webpack 5*, will introduce you to Webpack—specifically, Webpack version 5. It will include an overview of the central concepts around Webpack and how it is used.

`Chapter 2`, *Working with Modules and Code Splitting*, will elaborate on modules and code splitting, as well as some of the more salient and interesting aspects of Webpack 5 that are key to understanding Webpack.

`Chapter 3`, *Using Configurations and Options*, will look at the world of configuration, understanding its limitations and capabilities as well as where options can play their part.

`Chapter 4`, *APIs, Plugins, and Loaders*, will pry into the world of APIs, loaders, and plugins. These features of Webpack expound the capabilities of the platform, springboarding from configurations and options.

`Chapter 5`, *Libraries and Frameworks*, will discuss libraries and frameworks. Our examination of plugins, APIs, and loaders has revealed that sometimes, we don't want to use remote code such as libraries, but other times we do. Webpack generally deals with locally hosted code, but there are times when we might want to work with libraries. This provides a good transition to that topic.

`Chapter 6`, *Production, Integration, and Federated Modules*, will provide an in-depth account of this subject and will hopefully address any concerns you might have as a developer.

`Chapter 7`, *Debugging and Migration*, will discuss hot module replacement and live coding, and get to grips with some serious tutorials.

`Chapter 8`, *Authoring Tutorials and Live Coding Hacks*, will show you working examples of very simple, easy-to-apply, but bespoke practical lessons in Webpack 5, specifically Webpack 5 over earlier versions. There will be vanilla JavaScript tutorials as well as common frameworks, Vue.js will be a good one.

To get the most out of this book

You can find the code used in all of the chapters in this book at `https://github.com/PacktPublishing/Webpack-5-Up-and-Running`. To get the most out of this book, you will need the following:

- Fundamental knowledge of JavaScript.
- Ensure that you have the latest version of Webpack 5 installed.

- You will need the use of a command-line interface, such as Command Prompt or another command-line utility of your choice.
- You will need Node.js, the JavaScript runtime environment.
- Ensure that you've installed the latest version of Node.js (webpack 5 requires at least Node.js 10.13.0 (LTS)); otherwise, you may encounter a lot of issues.
- You will need `npm` installed on your local machine with administrator-level privileges. Webpack and Webpack 5 run in the Node.js environment, which is why we need its package manager—NPM.
- As of the time of writing, the most current release is Webpack version 5. Visit `https://webpack.js.org` to find the most current version for you.

Download the example code files

You can download the example code files for this book from your account at `www.packt.com`. If you purchased this book elsewhere, you can visit `www.packtpub.com/support` and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at `www.packt.com`.
2. Select the **Support** tab.
3. Click on **Code Downloads**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Webpack-5-Up-and-Running`. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "The following lines are a code snippet from the `package.json` file."

A block of code is set as follows:

```
"scripts": {
"build": "webpack --config webpack.config.js"
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
<!doctype html>
<html>
 <head>
 <title>Webpack - Test</title>
 <script src="https://unpkg.com/lodash@4.16.6"></script>
 </head>
 <body>
 <script src="./src/index.js"></script>
 </body>
</html>
```

Any command-line input or output is written as follows:

```
    npm install --save-dev webpack-cli
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Select **System info** from the **Administration** panel."

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at `customercare@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packtpub.com/support/errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `authors.packtpub.com`.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit `packt.com`.

1
Introduction to Webpack 5

This book is aimed at experienced JavaScript developers, designed to take you through the development and production of a specific example project through step-by-step processes. By the time you get to the end of this guide, you should be able to fully set up and deploy a working bundled application.

This chapter will introduce you to Webpack—specifically, Webpack version 5. It will include an overview of the central concepts around Webpack and how it is used.

This chapter is aimed at programmers who are new to Webpack and Webpack 5. The initial setup will be covered in this chapter, as well as an overview of the process, and you will be shown how to deploy your first bundled application.

The following topics will be covered in this chapter:

- The fundamentals of Webpack 5
- Setting up Webpack
- Creating a sample project

Technical requirements

You can find the code used in all of the chapters in this book at `https://github.com/PacktPublishing/Webpack-5-Up-and-Running`:

- To use this guide, you will need a fundamental knowledge of JavaScript.
- Ensure that you have the latest version of Webpack 5 installed.
- You will need the use of a command line, such as Command Prompt or another command-line utility of your choice.
- You will need Node.js, the JavaScript runtime environment.

- Ensure that you've installed the latest version of Node.js; otherwise, you may encounter a lot of issues.
- You will need `npm` installed on your local machine with administrator-level privileges. Webpack and Webpack 5 run in the Node.js environment, which is why we need its package manager—npm.
- As of the time of writing, the most current release is Webpack version 5. Visit `https://webpack.js.org` to find the most current version for you.

The fundamentals of Webpack 5

Essentially, Webpack is a module bundler for JavaScript applications. Webpack takes a series of JavaScript files, along with dependencies such as image files, which make up an application, and constructs something called a dependency graph. A dependency graph is a representation of how these files and dependencies are ordered and linked within an application and shows how the files interact with each other.

This dependency graph then forms a template that the bundler follows when taking all of the dependencies and files to compress them into a smaller set. Webpack is then able to bundle these files into a larger, but usually less-numerous, set of files. This eliminates problems such as unused code, repetitive code, and the need for rewriting. To some extent, the code can be formatted more succinctly.

Webpack recursively builds every module in your application, then packs all those modules into a small number of bundles. For the most part, a bundled application will contain a script that is ideal to be read by a program, such as a web browser, but too complicated for a programmer to use. The developer, therefore, will take a set of source files and make changes to this area of the program, then bundle this source into an output—a bundled application.

Bundling was originally intended to improve browser-reading performance, but it also has many other advantages. Once a set of source files is bundled by Webpack, it will usually follow a systematic and conventional filing structure. Errors within the code can halt a bundling operation; this book will instruct you on how to overcome these problems.

Now, let's explore the general concepts around Webpack 5.

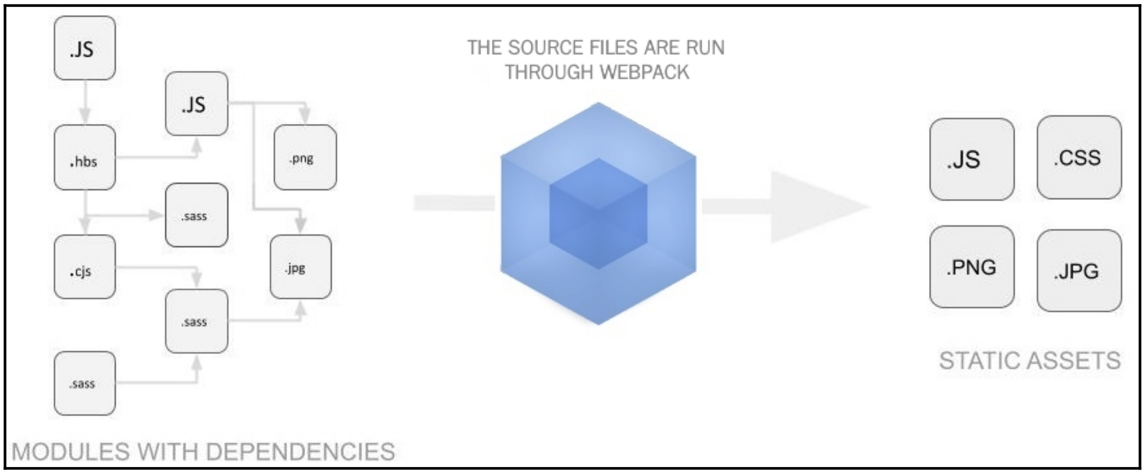
General concepts behind Webpack 5

Here, we will begin to understand the key concepts and the purpose of Webpack, rather than expect you to have any prior understanding of it. Bundling is carried out locally on a desktop using Node.js or `npm` and the **command-line interface** (**CLI**), usually Command Prompt.

Webpack is a build tool that puts all of your assets into a dependency graph. This includes JavaScript files, images, fonts, and **Cascading Style Sheets** (**CSS**). It will take **Sassy CSS** (**SCSS**) and TypeScript files and place them into CSS and JavaScript files, respectively. Webpack will only be able to do this when the code is compatible with the latter format.

When programming in JavaScript and other languages, the source code will often use a statement such as `require()`, which points one file to another. Webpack will detect this statement and determine the file that is needed as a dependency. This will decide how the files are processed in your final JavaScript bundle. This will also include replacing a URL path to a **content delivery network** (**CDN**)—which is, essentially, a network of proxy servers—with a local file.

The following diagram is a representation of the general purpose of Webpack, which is to take a set of files or dependencies and output the content in an optimized form:


.JS
.JS
.hbs
.png
.sass
.cjs
.jpg
.sass
.sass
MODULES WITH DEPENDENCIES
THE SOURCE FILES ARE RUN THROUGH WEBPACK
.JS
.CSS
.PNG
.JPG
STATIC ASSETS

Now, let's take a closer look at some of the terminology, which you may be unfamiliar with but can be considered common parlance when using Webpack.

Terminology

This section will cover the terminology used in Webpack 5. This will include native terms, as well as some more unusual acronyms:

- **Assets**: This is a term frequently used in Webpack to prevent conflations of concepts. It refers to image files, or even data or script files, collated by the software when producing a bundled application.
- **Bundle:** This refers to the application that is output once Webpack has compiled an application. This is an optimized version of the original or source application—the reasons for this will be discussed in detail in later chapters. The bundler will combine these files into one file, which makes unpicking and hacking very difficult. It also improves browser performance. It does this by ensuring that processors are kept to an optimal level and removing any coding structure that does not conform to a standard. This also encourages the developer to adopt conventions a lot more diligently. Should there be any insecure programming, these locations are more easily identified, isolated, and corrected.
- **SASS:** A version of CSS that has enhanced features. Webpack handles this code as it does CSS; however, it is a phrase that may come up and leave you stumped, so it is worth knowing about.
- **SCSS:** This is simply the name for the syntax version used to give SASS extra functionality. It is useful to know that Webpack is capable of transpiling both syntaxes.
- **Transpiling**: This is the process where Webpack 5 takes a set of input source code and changes it into a more optimized output distribution code. This is done by removing unused or duplicated code. Transpiling is used to convert one set of files into a simpler set. For instance, SCSS often contains script that can be easily stored inside a CSS file. You might also transpile SCSS to CSS, or TypeScript to JavaScript.
- **TypeScript**: For the uninitiated, TypeScript is a type of code that is similar to JavaScript in many ways. Browsers, for instance, most commonly run JavaScript, so it may be more appropriate to use JavaScript, where possible. Webpack 5 will transpile TypeScript into JavaScript whenever the former allows.
- **CDN:** A CDN is a network of proxy servers that provides high availability and high performance. Some examples are Google APIs, such as Google Fonts, and other similar tools that all JavaScript developers are, no doubt, familiar with.
- **Dependency graphs**: In Webpack 5, a dependency graph is a directed graph representing the dependency of several assets to each other. Webpack 5 maps a list of assets and dependencies itself and records how they depend on each other in an application. It uses this to derive an appropriate output file structure.

Though JavaScript is the entry point, Webpack appreciates that your other asset types—such as HTML, CSS, and SVG—each have dependencies of their own, which should be considered as part of the build process.

Webpack is comprised of **input** and **output**. The output can be made of one or more files. As well as bundling modules, Webpack can carry out a lot of functions on your files. The input refers to the original files when they are in their original structure, before bundling. The output refers to the resulting bundled files in their new and optimized filing structure. Inputs, therefore, are comprised of the source files and outputs can be comprised of development files or production files.

There is often a conflation between the terms input and output and source code and development code.

Source code refers to the original application before it is bundled. **Development code** refers to the application after it is placed in the Node.js environment and bundled in developmental mode. A more "tightly packed" version of the bundle is produced in production mode, but this version is difficult to work on. Therefore, the development code can be altered to some degree after it is bundled, which is very useful, for example, in a case where you are altering a database's connection configurations.

When working with Webpack 5, these phrases may present themselves and it is important that you don't get too confused by them.

Most other terminology will be explained as we encounter it, or it is so common that we assume you understand these terms if you are familiar with JavaScript.

That summarizes the bulk of the terminology you will come across when using Webpack. Now, we will explore how the software works.

How Webpack works

Webpack works by generating a dependency graph of assets in a set of source files, which it transpiles an optimized set of distribution files from. These source and distribution files contain source code and distribution code, respectively. This distributed code forms the output. Distribution is simply another name for the output or bundle.

Webpack begins by finding an entry point in the source files and from there, it builds a dependency graph. The selection of an entry point is optional in Webpack 5 and the choice of selection will alter the nature of the build process, either in terms of speed or output optimization.

Webpack 5 is capable of transforming, bundling, or packaging just about any resource or asset.

We have covered a good overview of how the software works; experienced users of previous Webpack versions may consider this overview rudimentary, so let's have a look at what is new in this current version.

What's new in Webpack 5?

The popular Webpack module bundler has undergone a massive update for the release of version 5. Webpack 5 offers massive performance improvements, more dynamic scalability, and basic backward compatibility.

Webpack 5 takes over from version 4, where backward compatibility wasn't always possible with many of the various loaders available, which were often more compatible with version 2, which meant that a developer would often encounter deprecation warnings in the command line if version 2 wasn't used. Webpack 5 has now addressed this issue.

Another big selling point of version 5 is federated modules. We will discuss this in more detail in a later `Chapter 6`, *Production, Integration, and Federated Modules*. However, to sum it up, federated modules are essentially a way for a bundled application to utilize and interact with modules and assets in remotely stored separate bundles.

The advantages of Webpack 5 are summarized as follows:

- Webpack 5 offers control over HTTP requests, which improves speed and performance, and also alleviates security concerns.
- Webpack 5 has some advantages over rival bundlers such as Browserify and systemjs—namely, speed. The build time directly depends on the configuration but it's faster than its nearest rivals.
- Little or no configuration is required to use Webpack 5, but you always have it as an option.
- It can be more complicated to use than alternatives, but this is mainly due to its versatility and scope and it is well worth overcoming.

- Webpack 5 has optimization plugins that remove unused code fantastically well. It also has many related features, such as tree shaking, which we will discuss in more detail later in this book.
- It is more flexible than Browserify, allowing the user to select more entry points and use different types of assets. It's also better, when it comes to speed and flexibility, for bundling large web applications and for single-page web applications.

Webpack is now considered an incredibly important tool for application development and web development, transforming the structure and optimizing the loading time for all of your web assets, such as HTML, JS, CSS, and images. Let's now get to grips with using Webpack practically. To do that, the first thing we will look at that may be new to you—if you have, perhaps, only worked with Vanilla JavaScript up to now—is modes.

Modes

Once you have understood the general concepts, the first thing you need to learn about when running a build is modes. Modes are central to how Webpack works and compiles projects, so it is best to cover this brief but important topic before we go any further.

Modes use the CLI, a process that we will cover in more detail later. If you are used to using Vanilla JavaScript, this may be new to you. However, rest assured it is not a complex topic to get your head around.

Webpack ships with two configuration files, which are as follows:

- **Development config**: This uses `webpack-dev-server` (hot reloading), debugging enabled, and so on.
- **Production config**: This will spit out an optimized, minimized (uglify JS), source-mapped bundle that is used in production environments.

Since the release of version 5, Webpack takes care of the mode features by default by simply adding a `mode` argument to the command. Webpack can't use `package.json` alone to find the mode to determine the correct build path.

Now that we have a grasp of the fundamentals, it's time to progress to the practical setup.

Setting up Webpack

This book follows the development of a sample project step by step, and I am sure you will find this a simple way to learn how to use Webpack 5.

Webpack 5 packages all its dependencies within the application you wish to bundle on a local machine. Theoretically, this can be done remotely, but to save any confusion for first-time users, I will emphasize the use of a local machine.

Installing packages locally is recommended for most projects. It makes things easier when upgrade or break changes are introduced.

We will begin with the `npm` installation. npm is the package manager that you will use with Webpack 5. Once this is installed on your local machine, you will be able to use the `npm` command using a CLI, such as Command Prompt.

Once you have installed `npm`, you can move on to the next step, which is to open your CLI. There are many to choose from, but for the sake of this tutorial, we will use Command Prompt.

Let's break this down step by step so that you can follow along:

1. Install the `npm` package manager, which you will use with Wepback 5.
2. Open the CLI (in this tutorial, we will be using Command Prompt) and type the following:

   ```
   mkdir webpack4 && cd webpack5
   npm init -y
   npm install webpack webpack-cli --save-dev
   ```

   Let's break down the code block. The preceding command will first create a new directory on your local machine, called `webpack5`. It will then identify the current directory (`cd`) as `webpack5`. This means any further commands made through the CLI will be made with respect to that directory. The next command is to initialize `npm`. A full list of these basic commands and what they mean can be found in the *Further reading* section at the end of this chapter. This section makes for some interesting reading and I'm sure you will learn something new. Then, we locally install Webpack and install `webpack-cli`—this is the tool used to run Webpack on the command line.

3. Next, install the latest release, or a specific version of Webpack, and run the following command. However, on the second line, replace `<version>` with the version of your choice, such as `5.00`:

```
npm install --save-dev webpack
npm install --save-dev webpack@<version>
```

4. The next command is `npm install`, which will install Webpack 5 in the directory and save the project in a development environment. It is important to note that there is a difference between the development and production environments (or modes):

```
npm install --save-dev webpack-cli
```

The following lines are a code snippet from the `package.json` file. We need these in the input files to generate a `webpack.config.js` file, which holds the configuration information for your Webpack bundle.

5. We must take care to ensure that the `package.json` file is coded as follows:

```
"scripts": {
"build": "webpack --config webpack.config.js"
}
```

When using Webpack 5, you can access its binary version by running `npx webpack` in the CLI.

We should also decide which type of installation we need; any re-installation will overwrite the previous one, so don't worry if you have already followed the preceding steps.

6. Let's do that installation now, if applicable.

There are two types of installation:

- **Global**: A global installation will lockdown your installation to a specific version of Webpack.
  The following `npm` installation will make Webpack available globally:

```
npm install --global webpack
```

- **Local**: A local installation will allow you to run Webpack in the project directory. This needs to be done via the `npm` script:

```
npm install webpack --save-dev
```

You will need to carry out all of the preceding steps every time you begin a new project on a new local machine. Once you have completed the installation, it is time to revert your attention to building a project.

Creating a sample project

Now, we'll create an experimental project with the following directory structure, files, and their contents.

The following code block refers to a folder on your local machine. It illustrates the format and naming conventions typically used in Webpack. You should follow this format to ensure your project aligns with this tutorial, as follows:

1. Begin by setting up the **project tree**:

```
webpack5-demo
 |- package.json
  |- index.html
  |- /src
  |- index.js
```

The project tree shows us the files that we will work on.

2. Let's now take a closer look at the index files as they will be key to our frontend, starting with `src/index.js`:

```
function component() {
 let element = document.createElement('div');
// Lodash, currently included via a script, is required for this
// line to work
 element.innerHTML = _.join(['Testing', 'webpack'], ' ');
 return element;
}
document.body.appendChild(component());
```

`index.js` contains our JS. The `index.html` file that follows is our user's frontend.

3. It will also need setting up, so let's open and edit `index.html`:

```
<!doctype html>
<html>
 <head>
 <title>Webpack - Test</title>
 <script src="https://unpkg.com/lodash@4.16.6"></script>
 </head>
 <body>
 <script src="./src/index.js"></script>
 </body>
</html>
```

Note the preceding `<script src="https://unpkg.com/lodash@4.16.6">` tag. This refers to the use of the `lodash` library. The `index.js` file (not the `index.html` file) requires this library to be called. Webpack will take whatever modules it needs from the library and use them to build a dependency graph for the bundle.

Lodash is a JavaScript library that provides functional programming tasks. It was released under the MIT license and essentially makes things easier when working with numbers, arrays, strings, and objects.

Something to be aware of is that if it is not made clear that your code depends on an external library, the application will not function properly. For example, dependencies could be missing or included in the wrong order. Conversely, the browser will download unnecessary code if a dependency is included but not used.

We can manage these scripts by using Webpack 5.

4. You will also need to adjust your `package.json` file to mark your package as private, as well as removing the main entry point. This is to prevent accidentally publishing your code:

```
{
 "name": "webpack5",
 "version": "1.0.0",
 "description": "",
 "private": true,
 "main": "index.js",
 "scripts": {
 "test": "echo \"Error: no test specified\" && exit 1"
 },
 "keywords": [],
 "author": "",
"license": "ISC",
"devDependencies": {
"webpack": "^5.0.0",
"webpack-cli": "^3.1.2"
},
"dependencies": {}
}
```

You can see how to make these alterations from the bold text in the preceding code. Note that our entry point will be set as `index.js`. This is the first file that Webpack will read when beginning a bundle compilation (see the previous definition of a dependency graph).

If you want to learn more about the `package.json` file, go to `https://docs.npmjs.com/getting-started/`, which gives you information about `npm`.

We have now finished with the source code for the first demonstration application bundle. This constitutes the input or source files that we will now run through Webpack to produce our first bundled application.

Bundling your first project

Web packing simply means bundling the project. It is the essence of Webpack and starting with this very simple introduction is an excellent way to begin learning about the application.

Firstly, we need to separate the source code from our distribution code by altering our directory structure slightly. This source code is used to write and edit and the distribution code is the minimized and optimized bundle that is the result of our build process.

We will now go through each step for building our first project in detail:

1. We will begin by structuring the project and the directories. First, note the `/src` and `/dist` terms; they refer to the source code and distribution code, respectively:

```
webpack5-demo
|- package.json
  |- /dist
  |- index.html
  |- index.js
|- /src
|- index.js
```

2. To bundle the `lodash` dependency with `index.js`, we need to install the library locally:

   ```
   npm install --save lodash
   ```

   When installing a package that will be bundled to your production bundle, you should use the following command:

   ```
   npm install --save
   ```

   If you're installing a package for development purposes (for example, a linter, testing libraries, and so on), you should use the following command:

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

3. Now, let's import `lodash` into our script using `src/main.js`:

   ```
   import_ from 'lodash';

   function component() {
   let element = document.createElement('div');
    // Lodash, currently included via a script, is required for this
   // line to work
   element.innerHTML = _.join(['Hello', 'Webpack'], ' ');
   return element;
   }
   document.body.appendChild(component());
   ```

4. Next, update your `dist/index.html` file. We will remove the inclusion of the `lodash` library.

   This is done because we will be installing the library locally for bundling and no longer need to make a remote call to the library:

   ```
   <!doctype html>
   <html>
   <head>
   <title>Getting Started</title>
     <script src="https://unpkg.com/lodash@4.16.6"></script>
     //If you see the above line, please remove it.
   </head>
   <body>
     <script src="main.js"></script>
   </body>
   </html>
   ```

5. Next, we will use the command line to run `npx webpack`. The `npx` command ships with Node 8.2/npm 5.0.0 or higher and runs the Webpack binary (`./node_modules/.bin/webpack`). This will take our script at `src/index.js` as the entry point and will generate `dist/main.js` as the output:

```
npx webpack
...
Built at: 14/03/2019 11:50:07
Asset Size Chunks Chunk Names
main.js 70.4 KiB 0 [emitted] main
...
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/concepts/mode/
```

   If there are no errors, the build can be considered successful.

   Note that a warning is not considered an error. The warning is simply shown because no mode has yet been set.

   I wouldn't be concerned about this as Webpack will default to production mode. We will handle the setting of modes later in this guide.

6. You should see the following text when you open `index.html` in your browser:

```
Testing Webpack5
```

Huzzah—we have completed our first application bundle and I bet you're very proud of yourself! This was a fundamental step to begin with; we will move on to more complex elements of Webpack in later chapters and begin applying them to existing projects that need bundling.

Summary

To summarize, Webpack 5 is an incredibly versatile bundler that uses almost every conceivable method to optimize the size of applications and improve the overall performance. It is very worthwhile getting to grips with it and this guide will show you everything you need to know to do so.

You should now understand the basic concepts behind Webpack, as well as the fundamental terminology. You should also now know how to install the prerequisites, such as Node.js, and set up and deploy—as well as make—your first bundle using the command line.

In the next chapter, we will elaborate on modules and code splitting, as well as some of the more salient and interesting aspects of Webpack 5 that are key to understanding Webpack.

Questions

The following are a series of questions related to this chapter that you should try to answer to aid your learning. The answers can be found under the *Assessments* section in the back matter of this book:

1. What is Webpack?
2. What is a bundle in Webpack?
3. What is the latest version of Webpack, according to this guide?
4. Which environment does Webpack work in?
5. What is a dependency graph?
6. When bundling, what entry is missing from the following command:
   `npm --save lodash`
7. What is the name of the package manager that we use with Webpack 5?
8. How would you remove the `lodash` library using the command line?
9. What is the difference between source code and distribution code when working with Webpack 5?
10. When setting up your project, why might you adjust the `package.json` file?

2 Working with Modules and Code Splitting

This chapter will explore modules and code splitting in Webpack 5. Modules are a way of structuring content so that code is sorted in sections by function. Code splitting is the method Webpack uses to build these modules automatically; it will parcel off code from your project into modules that best suit the functionality and structure of the completed project.

The topics covered in this chapter are as follows:

- Explaining modules
- Understanding code splitting
- Prefetching and preloading modules
- Best practices

Explaining modules

Webpack works with elements called modules. It uses these modules to build a dependency graph.

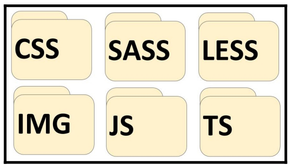
Modules are sections of code that deal with a related function; structuring your project according to a modular build will improve functionality. For instance, only code related to pertinent actions will need to be run, when compared to the need to run whole sections of unrelated code when the project is not built with modules.

With that being said, the next thing to understand is the specific function of modules, which will be discussed in the following section.

The function of modules

A module is a set of pieces of code: for example, code of a similar language has a common function—that is to say, it is part of the same function or operation within an application.

Often, modules in Webpack 5 are grouped according to the scripting language used, as follows:



The preceding diagram should help illustrate what most people see when exploring the contents of a Webpack build.

An application is then divided into modules and assets. As we explained in `Chapter 1`, *Introduction to Webpack 5*, an asset is essentially images and videos that are not considered scripted by a developer. The directory structure is then subdivided into these modules, usually in a directory of their own.

Dividing an application into modules will naturally make the process of debugging easier. This will also aid us in verification and testing in general.

Building applications this way ensures that a boundary is made between well-written code and more-dubiously written code. Naturally, this helps with directory navigation, as each module has a defined purpose.

Many platforms use modules and it is a term that you will no doubt be used to if you work in web development in general. However, each platform differs slightly.

Webpack 5 forms these modules according to how it expresses the dependencies of the module. Here are a few examples of how Webpack 5 expresses them:

- Through a **2015 ECMAScript** `import` statement
- Through a **CommonJS** `require()` statement
- Through an **asynchronous module definition** (**ASM**) `define` and `require` statement
- Through an image URL in a stylesheet
- Through an `@import` statement in a stylesheet

In summary, modular code makes things a lot easier and understanding how Webpack expresses dependencies will help you understand how you should compile your code. From here, the natural next step is to look at the supported module types and how loaders work with them.

Supported module languages and loaders

To make sure Webpack 5 supports these modules, they must be written in a programming language that can be understood and processed. Webpack 5 does this by using something called loaders.

Loaders make Webpack truly stand out over rival bundlers. In simple terms, a loader tells Webpack how to process code that is not JavaScript or other predefined code that Webpack automatically understands, such as JSON or HTML. Webpack 5 will then include this processed code as dependencies in your bundle.

Webpack 5 has a community of developers, referred to as the Webpack community, who have built these loaders. These loaders currently support a large number of languages and processors; some examples are as follows:

- **TypeScript**
- **SASS**
- **LESS**
- **C++**
- **Babel**
- **Bootstrap**

For a full list of available loaders, see the *Further reading* section toward the end of this chapter.

Being part of the Webpack community means you can write your loaders! This is something worth considering as it may be the best way to meet the requirements of your project.

There are many more loaders available from the Webpack community. The use of loaders means Webpack 5 can be described as a dynamic platform that allows the customization of just about any technology stack. In this chapter, we will begin to use loaders proactively as part of some example use cases that you can practice coding yourself.

During your development, you may come across the term **encapsulation**, especially when working with modules and loaders.

To understand encapsulation, you first need to understand that software can sometimes be developed independently until a requirement for interplay presents itself. For software to work together within a project, a dependency must be created between the two technology stacks. This is what is meant by the term encapsulation.

Encapsulation is a simple topic to outline; however, the next area of modular coding concerns resolution. It is a wider subject and has, therefore, been detailed as part of its own subsection.

The enabling of a new asset module type is an experimental feature shipped with v5. The asset module type is similar to the `file-loader`, `url-loader`, or `raw-loader` (`experiments.asset` since alpha.19) data URLs and options related to that have been supported since beta.8.

Module resolution

Module resolution takes place via the use of a resolver. A resolver assists you in locating a module by its absolute path—the path to a module that is universally used throughout a project.

Note that a module can act as a dependency of another module, such as the following:

```
import example from 'path/to/module';
```

Regardless of whether the dependency module is from another library (not the resolver itself) or the application itself, the resolver will help to find the module code required to include in the bundle. Webpack 5 can also use `enhance-resolve` to resolve paths while bundling.

The resolving rules for Webpack 5 mean that, using the `enhanced-resolve` method, Webpack 5 can resolve three kinds of file paths:

- Absolute paths
- Relative paths
- Module paths

The following sections will detail the meaning of each file path and there will be an example for each. This will become important later as we begin to build on our project bundle.

Absolute paths

For the uninitiated, an absolute path refers to the file path and the location of a file that is common to all files and assets that your project uses. This common location is sometimes called the `home` or `root` directory. Here is a command-line location by way of example:

```
import 'C:\\Users\\project\\file';
```

The preceding line is an example of an absolute path. The term **absolute** is something every JavaScript developer should be familiar with. It relates to the location of an object file or directory in a path that is universal throughout the system.

If we already have the absolute path, as in the preceding line, no further resolution is required.

Relative paths

A relative path refers to the location of an object file or directory to another location. In this case, it is the location of the `context` directory—the current and working location where development is carried out:

```
import '../src/file';
```

In the preceding example, the directory of the resource file is considered the `context` directory. The resource file refers to the file that the `import()` statement, the `require()` statement, or the call to an external file occurs.

In this case, the relative path is joined to the context directory path, which then produces an absolute path.

Module paths

A module path is something that not all JavaScript developers may be used to. With Webpack, it refers to a location relative to a module. In the following code snippet, `module` would be co-opted for the name of any specific module name you wish to use—the name of an existing module in your project, for example:

```
import 'module/sub-directory/file';
```

Webpack 5 searches all the directories for modules that are specified in the `resolve.module` directive. An alias can be created for each original module file path using the `resolve.alias` configuration. Using this method, Webpack 5 checks the file path and whether it points to a file or directory.

Webpack 5 has an option called `resolve.extension`. If a path does not have a file extension, this resolver will indicate to Webpack which extensions can be used for resolution. These might include `.js`, `.jsx`, or similar extensions.

In the event of a file path not pointing to a file but only to a directory, Webpack 5 searches the directory for a `package.json` file. Webpack 5 then uses the fields specified in the `resove.main` field's configuration to search for fields contained in `package.json` and, from this, determines the correct contextual file path to use.

If there is no `package.json` file present in the directory, or if the main fields do not return a valid path, Webpack 5 simply searches for filenames specified in the `resolve.main` configuration.

File extensions are resolved similarly, but using the `resolve.extension` option.

We have, so far, covered modules, path resolution, supported languages, and loaders. The next crucial thing to understand is code splitting—what it is and how Webpack utilizes it to form both its modules and general output.

Understanding code splitting

Code splitting allows the user to **split code** into various bundles that can then be loaded on-demand or in parallel. The developers of Webpack consider this "*one of the most compelling features of Webpack*" (`Webpack.js.org`).

Code splitting has two key advantages—the process can be used to achieve smaller bundles and to control the priority of resource loading. This can lead to an improvement in loading time.

There are three general code-splitting approaches available in Webpack 5:

- **Entry points**: This manually splits code using an entry point configuration.
- **Prevent duplication**: This approach uses `SplitChunksPlugin` to run a process called **dedupe**, which splits code into groups of modules called **chunks**.
- **Dynamic imports**: This approach uses inline functions to make **calls** to split code within modules.

A chunk refers to a group of modules. This is a term used by Webpack and is not frequently encountered on other platforms.

dedupe is a Python library that uses machine learning to perform matching, **deduplication**, and entity resolution quickly. It helps remove duplicate entries from a spreadsheet of names and addresses.

With these three approaches outlined, we can now discuss each one in detail in the following sections. Let's begin with entry points.

Entry points

Using entry points is probably the easiest way of performing code splitting. It is a manual operation and is, therefore, not automated like other methods.

We will now look at the development of splitting one module from the main bundle. To do this, we will begin with some practical work. From there, we will go over the concepts of duplication and dynamic imports.

We will now return to the project we were working on in `Chapter 1`, *Introduction to Webpack 5*. This time, we will utilize what we have learned so far in this chapter.

First, create a directory to work in. In this case, we are using the directory name we used in the last chapter. It might be a good idea to follow this same convention and, that way, you will be able to follow the course of the project's development as you continue through this book.

In the following example, we will do the following:

1. Organize a project folder structure to start a project that shows how entry points work. You should build this set of directories in your practice project directory. This is done in the same way as creating folders on your desktop. For the sake of this example, we will call this folder `webpack5-demo` (but you can choose any name you want):

```
package.json
webpack.config.js
/dist
/src
 index.js
/node_modules
/node_modules/another-module.js
```

2. Be sure to add the last line of text (in bold), if it is missing from the code you are using. This can be done on a command line; if that's what you decide to use, please refer to `Chapter 1`, *Introduction to Webpack 5*, for guidance. You may have noticed the inclusion of `another-module.js`. You might not find this a typical build but you will need to include this for our example.

Ultimately, you can name the project anything you like but, for the sake of following this practice project, you should use the same naming convention used up to now to prevent confusion.

To follow this project development, using your **integrated development environment** (**IDE**) or notepad, you should create each of the preceding files and folders. The **/** character indicates a folder. Note the `another-module.js` file; this rests in the `/node_modules` directory.

We will now edit and compile a build, beginning with the `another-module.js` file.

3. Open `another-module.js` in your IDE of choice or a notepad:

```
import _ from 'lodash';
console.log(
  _.join(['Another', 'module', 'loaded!'], ' ')
 );

// webpack.config.js

 const path = require('path');
 module.exports = {
   mode: 'development',
   entry: {
     index: './src/index.js',
     another: './src/another-module.js'
 },
 output: {
   filename: '[name].bundle.js',
   path: path.resolve(__dirname, 'dist')
  }
 };
```

The file essentially imports `lodash`, ensuring the module that is loaded is recorded in the console log, setting the Webpack build mode to development, and setting entry points that Webpack begins mapping the assets in the application for bundling through and sets an output bundle name and location.

4. Now, run a build with `npm` by entering the location of the context directory (the one you are developing in) in the command line and type the following:

   ```
   npm run build
   ```

   This is all you need to produce a bundle output or development application.

5. Then, check for successful compilation. When a build is run in your command line, you should see the following message:

   ```
   ...
    Asset Size Chunks Chunk Names
    another.bundle.js 550 KiB another [emitted] another
    index.bundle.js 550 KiB index [emitted] index
    Entrypoint index = index.bundle.js
    Entrypoint another = another.bundle.js
    ...
   ```

Success! However, some potential problems might occur when using entry points that a developer should be conscious of:

- If there are duplicated modules between entry chunks, they will be included in both bundles.

For our example, as `lodash` is also imported as part of the `./src/index.js` file in the project directory, it will be duplicated in both bundles. This duplication can be removed by using `SplitChunksPlugin`.

- They can't be used to dynamically split code using the programming logic of the application.

Now, we will cover preventing duplication.

Preventing duplication with SplitChunksPlugin

`SplitChunksPlugin` allows the extraction of common dependencies into entry chunks, either existing or new. In the following walk-through, this method will be used to deduplicate the `lodash` dependency from the preceding example.

The following is a code snippet from the `webpack.config.js` file, found in the preceding example's project directory. This example shows the configuration options needed to use the plugin:

1. We will begin by ensuring that our configuration is coded to the same configuration as in the preceding example:

```
const path = require('path');
module.exports = {
  mode: 'development',
  entry: {
    index: './src/index.js',
    another: './src/another-module.js'
 },
 output: {
   filename: '[name].bundle.js',
   path: path.resolve(__dirname, 'dist')
 },
 optimization: {
   splitChunks: {
     chunks: 'all'
   }
  }
 };
```

   Using the `optimization.splitChunks` configuration, the duplicate dependency should now be removed from `index.bundle.js` and `another.bundle.js`. `lodash` has been separated into a separate chunk and the main bundle.

2. Next, perform `npm run build`:

```
...
Asset Size Chunks Chunk Names
another.bundle.js 5.95 KiB another [emitted] another
index.bundle.js 5.89 KiB index [emitted] index
vendors~another~index.bundle.js 547 KiB vendors~another~index
[emitted]    vendors~another~index
Entrypoint index = vendors~another~index.bundle.js index.bundle.js
Entrypoint another = vendors~another~index.bundle.js
another.bundle.js
...
```

There are other community-developed loaders and plugins that can be used to split code. Some of the more notable examples are as follows:

- `bundle-loader`: Used to split code and lazy-load the resulting bundles
- `promise-loader`: Similar to `bundle-loader` but uses promises
- `mini-css-extract-plugin`: Useful for splitting CSS from the main application

Now, with a firm understanding of how duplication can be prevented, we will move on to a more difficult topic—dynamic imports.

Dynamic imports

Dynamic imports are, essentially, on-demand imports on Webpack. If you have already bundled a lot of code but need to patch more to it, the dynamic import method will come to the rescue. This also includes dynamic code splitting, which, as it sounds, means splitting code and optimizing it after a bundle has been built.

Webpack 5 supports two methods of doing this:

- The first method uses the `import()` syntax, which conforms to the dynamic import proposal for ECMAScript.
- The second is the `webpack-specific` approach, which uses the `require.ensure` method (this is a legacy method).

The following is an example of the first approach; the goal is to demonstrate a modern method of using dynamic imports, which will be more common on recent projects.

The `import()` call is an internal call to promises. A **promise** refers to the information returned from a loader.

When using `import()` with older browsers, use a `polyfill` function—such as `es6-promise` or `promise-polyfill`—to **shim promise**. `shim-loader` is a loader that transforms code so that it works in the Webpack 5 environment; this works similarly to doing this manually with `imports-loader` and `exports-loader`.

The next step is to remove any surplus entries in the configuration file, which includes the `optmization.splitChunks` reference, as it will not be needed in the following demonstration:

1. Now, open the `webpack.config.js` file and make the following entries:

```
const path = require('path');
module.exports = {
 mode: 'development',
 entry: {
   index: './src/index.js'
   index: './src/index.js',
 },
 output: {
   filename: '[name].bundle.js',
   chunkFilename: '[name].bundle.js',
   path: path.resolve(__dirname, 'dist')
 },
 };
```

Note the use of `chunkFilename`, which determines the name of non-entry chunk files.

The preceding configuration was needed to prepare your project to use dynamic imports. Be sure to remove the text in bold as you may see this when working with the same code, as before.

Jumping back into the project, we need to update it with instructions to remove unused files.

You may have already set up your practice directory; however, it is recommended that you start with a fresh set of directories that doesn't contain any experimental code.

The following demonstration will use dynamic importing to separate a chunk, instead of the static importing of `lodash`.

2. Open the `index.js` file and ensure the following entries are made:

```
function getComponent() {
  return import(/* webpackChunkName: "lodash" */ 'lodash').then((
      { default: _ }) => {
    var element = document.createElement('div');
```

```
      element.innerHTML = _.join(['Hello', 'Webpack'], ' ');

      return element;

    }).catch(error => 'An error occurred while loading
       the component');
   }

    getComponent().then(component => {
      document.body.appendChild(component);
    })
```

When importing a `CommonJS` module, this import will not resolve the value of `module.exports`; instead, an artificial namespace object will be created. Therefore, we need a default when importing.

The use of `webpackChunkName` in the comment will cause our separate bundle to be named `lodash.bundle.js`, instead of just `[your id here].bundle.js`. For more information on `webpackChunkName` and the other available options, see the `import()` documentation.

If Webpack is now run, `lodash` will separate into a new bundle.

3. `npm run build` can be run using your **command-line interface** (**CLI**). In your CLI utility, type the following:

```
npm run build
```

When a build is run, you should see the following message:

```
...
 Asset Size Chunks Chunk Names
 index.bundle.js 7.88 KiB index [emitted] index
 vendors~lodash.bundle.js 547 KiB vendors~lodash [emitted]
vendors~lodash
 Entrypoint index = index.bundle.js
 ...
```

`import()` can be used with asynchronous functions as it returns a promise. This requires the use of a preprocessor, such as the `syntax-dynamic-import` Babel plugin.

4. Using `src/index.js`, make the following amendments to show how the code can be simplified:

```
async function getComponent() {
 'lodash').then(({ default: _ }) => {
const element = document.createElement('div');
const { default: _ } = await import(/* webpackChunkName: "lodash"
*/ 'lodash');

element.innerHTML = _.join(['Hello', 'webpack'], ' ');

return element;
}

  getComponent().then(component => {
    document.body.appendChild(component);
  });
```

The preceding example uses the same file that we used in the *Dynamic imports* section. We have turned multi-line code into single lines, replaced a returning function with asynchronous code, expediting our coding practice. You will see that it is now much simpler than the earlier code—it uses the same file, `src/index.js`, and achieves the same thing.

We often simplify code to help with loading times. Another key way of improving browsing speed is caching.

Caching

Before we finish this section on code splitting, we will go over caching. Caching is related to the previous processes and is no doubt something that will come up during programming. For the uninitiated, caching is the method of storing previously computed data to allow it to be served faster. It also relates to the following section on prefetching and preloading, methods that govern how memory is used.

Learning about caching will ensure you know how to split code more effectively. In the following example, we will see how it's done. In Webpack, caching is done by something called **filename hashing** (when a computer traces the location of a file recursively) and, specifically, the hashing of the output bundle:

```
module.exports = {
  entry: './src/index.js',
  plugins: [
   // new CleanWebpackPlugin(['dist/*']) for < v2 versions
```

```
         of CleanWebpackPlugin
       new CleanWebpackPlugin(),
       new HtmlWebpackPlugin({
         title: 'Output Management',
         title: 'Caching',
       }),
     ],
     output: {
      filename: 'bundle.js',
      filename: '[name].[contenthash].js',
      path: path.resolve(__dirname, 'dist'),
      },
    };
```

Note the `output` key handler in the preceding code block; within the parentheses, you will see the output `bundle.js` filename and below that is the inline element we refer to as the hash. You should substitute the bracketed terms with your preferences. This method produces an alternative output that is only updated when the content updates and serves as our cache resource.

Every filesystem access is cached so that multiple parallel or serial requests to the same file occur faster. In `watch` mode, only modified files are evicted from the cache. If `watch` mode is turned off, then the cache is purged before every compilation.

This leads us to our next section, which also relates to imports—prefetching and preloading.

Prefetching and preloading modules

Webpack 5 can output a **resource hint** when using inline directives and declaring imports. It will give the browser the following commands:

- `preload` (may be needed during the current navigation)
- `prefetch` (may be needed for future navigation)

The terms "current" and "future" may be confusing, but they essentially refer to the fact that `prefetch` loads content before the user needs it, which, in a way, loads and queues content ahead of time. This is a simple definition—a full explanation will follow—but in general, you can see the advantages and disadvantages in terms of memory usage and efficiency of user experience.

One thing to be aware of is that prefetching doesn't work for **Web Assembly** (**WASM**) yet in Webpack 5.

This simple `prefetch` example can have a `HomePage` component that renders a `LoginButton` component, which, upon being clicked, loads a `LoginModal` component.

The `LoginButton` file will need to be created; follow these instructions in `LoginButton.js` to do this:

```
import(/* webpackPrefetch: true */ 'LoginModal');
```

The preceding code will result in the following code snippet being appended to the header of the page:

```
<linkrel="prefetch" href="login-modal-chunk.js">
```

This will instruct the browser, when idle, to prefetch the **`login-modal-chunk.js`** file.

The `preload` directive has many differences when compared to `prefetch`:

- Chunks using the `preload` directive load in parallel to their parent chunk, whereas a prefetched chunk starts after the parent chunk finishes loading.
- Chunks must be instantly requested by the parent chunk when preloaded, whereas a prefetched chunk can be used at any time.
- Chunks using the `preload` directive are instantly downloaded when called. A prefetched chunk is downloaded while the browser is idle.
- Simple `preload` directives can have components that always depend on libraries that should be in a separate chunk.

The choice of whether to use `preload` or `prefetch` depends largely on context; you will discover more on how that might apply to you as the tutorial progresses.

You should use `prefetch` or `preload`, depending on how the preceding points best suit your development needs. This largely depends on the complexity of the project and, ultimately, is a judgment call made by the developers.

The following example suggests an imagined component, `ChartComponent`, in `ChartComponent.js`, which requires a library that we will call `ChartingLibrary`. It instantly imports the library on demand and displays `LoadingIndicator` when rendered:

```
import(/* webpackPreload: true */ 'ChartingLibrary');
```

When `ChartComponent` is requested, `charting-library-chunk` is also requested through `<link rel="preload">`.

Assuming `page-chunk` finishes loading faster, the page will be displayed with `LoadingIndicator` until `charting-library-chunk` finishes loading. This will give a loading time improvement since it only needs one round process instead of two. This is especially true in high-latency environments (where delays often occur in data processing networks).

Using `webpackPreload` incorrectly can harm the performance, so be mindful when using it.

One feature added with version 5 that is useful and related to fetching is the top-level await, a feature that enables modules to act as big async functions. That means they will be processed as code asynchronously. With top-level await, **ECMAScript Modules** (**ESMs**) can await resources, causing other modules that import them to wait before they start evaluating the body.

You should now understand the purposes of `prefetch` and `preload` and how their usage affects performance if done incorrectly. A decision on their use will depend largely on how you wish your application to perform. The best approach is to make your decision on their use after conducting a formal bundle analysis, which we will discuss in the next section.

Best practices

As with all programming, there are best practices that ensure the most optimum delivery. This is also an excellent way to conclude this chapter. If best practices are followed, a developer can protect their application from security breaches and hacking, poor performance, and difficulties when working collaboratively in a team, or when future development is needed by new developers, future-proofing the build. This latter reason speaks to the product owner or project manager more than a development team.

When it comes to Webpack, the foremost areas of salience here would be bundle analysis and code linting.

Bundle analysis

Once you start splitting code, it can be useful to analyze the output and check where modules have ended up. It's important to get the most out of bundling, so a formal procedure for bundle analysis can be considered as fundamental, as well as browser and security testing. The official analyze tool is recommended. There are also some other options:

- `webpack-bundle-analyzer`: This is a plugin and CLI utility that represents bundle content as a convenient interactive **treemap**, where zooming options are available.
- `webpack-bundle-optimize-helper`: This tool will analyze your bundle and give suggestions to reduce your bundle size.
- `webpack-visualizer`: This is used to visually analyze bundles to see which modules are taking up too much space and which might be duplicated.
- `webpack-chart`: This offers an interactive pie chart used for Webpack stats.

Treemapping is a method used to display hierarchical data using nested figures, usually rectangles.

All the previously mentioned tools will help with optimization, which is the primary purpose of Webpack.

Code linting

Another way that the application can be improved is through the removal of unwanted code. When automated, this is often called tree shaking, which we will discuss in later chapters. When this is done manually, it is referred to as code linting. A definition is probably in order as it is a phrase that is not often encountered in programming.

Code linting is the process of removing unwanted or surplus code, just like removing lint from a blazer. This could include unused coding artifacts, erroneous code, or anything else that's unwanted. Webpack uses an automated process to do this when integrating with task runners such as **Gulp**. This is discussed in the following chapter, `Chapter 6`, *Production, Integrations, and Federated Modules*.

If you follow these procedures, then there can be little doubt that your application will perform at its very best. Code splitting and modular programming are central to Webpack and a firm understanding is required to prevent you from getting lost as the complexity of bundling projects advances through this guide.

Summary

This chapter has followed the demonstration of various code-splitting practices, including code chunks and dynamic imports. You will now have a sound base of knowledge to carry out code splitting and use modules. These are fundamental features of Webpack, so a solid grounding is necessary.

Code splitting and modules are a matter of structural necessity for Webpack applications. Code chunks and dynamic imports will be more important on specialist tasks that require heavy-duty programming.

You were taken through prefetching modules and bundle analysis—important procedures that are needed to clearly understand the following chapter, where we will look at the world of configuration, understanding its limitations and capabilities as well as where options play their part.

These concepts are important as configuration is central to Webpack development and a matter of day-to-day programming. Options become more important when it comes to production and you need your project to operate correctly.

To test your skills, try the following quiz and see whether your understanding of the topics covered in this chapter is up to scratch.

Questions

We will conclude this chapter with a set of questions to test your knowledge. The answers to these questions can be found in the back of this book, in the *Assessment* section:

1. What is the difference between code splitting and modular programming?
2. What is a code chunk?
3. How do dynamic imports differ from entry points?
4. How are `preload` directives preferable to `prefetch` directives?
5. What is meant by code linting?

6. What is meant by the term promise?
7. How does `SplitChunksPlugin` prevent duplication?
8. What does the `webpack-bundle-optimize-helper` tool offer?
9. What does the `webpack-chart` plugin do?
10. What is treemapping?

Further reading

For a full list of loaders, go to `https://github.com/webpack-contrib/awesome-webpack`.

3
Using Configurations and Options

This chapter will include practical usage of configurations and options, and how these interrelate and play their part in any given build. It will also elaborate on output management—that is to say, the output of the bundling process and asset management in terms of the assets made part of the dependency graph. This will cover subtopics such as file placement and file structure.

Modules are used to combat the nature of JavaScript to have global functions. Webpack works alongside these modules and isolates the implied global nature of variables and functions.

Configurations and options are necessary to get the best out of Webpack. Each project is bespoke, therefore each project will require specific tailoring of its parameters. This chapter will explore in detail the exact nature of both topics, the limitations of each, and when to utilize them.

The following topics are discussed in this chapter:

- Understanding configurations
- Understanding asset management
- Understanding output management
- Exploring Webpack 5 options

Understanding configurations

The configuration is done in Webpack through the use of a configuration file, usually `webpack.config.js`, except in special circumstances where there can be more than one—or another—file assigned to this task. In the case of `webpack.config.js`, it is a JavaScript file that should be amended to alter the configuration settings of any particular project.

On startup, Webpack and Webpack 5 do not require a configuration file, but the software does recognize `src/index` as the default project input. It will also output the results of the build to a location named `dist/main.js`. This output will be "minified" and optimized for production.

Minified, or *minification,* simply refers to one of Webpack's main functions: to reduce the amount of code used to a minimal amount. This is done by eliminating duplicate, erroneous, or surplus code.

Typically, however, a Webpack project will need to have its default configuration changed. The default configuration is how Webpack works without any loaders or special parameters assigned to it, such as described in `Chapter 1`, *Introduction to Webpack 5*, in the *How Webpack works* subsection. This is done through the use of the configuration file. The developer should create a file named `webpack.config.js` and place this in the root folder of the project. This file will be automatically detected and read by Webpack.

Let's begin our discussion by exploring the use of multiple configuration files.

Using different configuration files

Webpack 5 gives the option of using different configuration files, depending on the situation. Not only that, but the files being used can be altered using a command-line utility. A typical situation in which you might do that is when working with multiple bundles inside one project—more on that topic later on in the guide. The following code snippet shows how the developer can alter the configuration file in use. In this example, a file is directed to a named `package.json` file, which is a common file that Webpack uses regularly. This technique is referred to as a *config flag*:

```
"scripts": {
  "build": "webpack --config example.config.js" }
```

Please note that Webpack 5 also allows for custom configuration, and, as explained in `Chapter 1`, *Introduction to Webpack 5*, this is a salient advantage of using Webpack 5. This is done via the use of custom configuration files. This differs from options, as these variables are not set using the **command-line interface** (**CLI**).

Working with options

The term *options* in Webpack refers to settings made from the command line rather than the configuration, which is done through the altering of configuration script.

In the following example, we will begin by altering the configuration file, simply to lay the groundwork for our options tutorial.

Throughout the following configuration, Node's **path module** is used and prefixed with the `_dirname` global variable. Node's path module is simply the utility that Node uses for working with file or directory paths. There may be file path problems when working between operating systems, and this prevents those problems from occurring and also ensures that relative paths work correctly.

The file concerned in the example is called `webpack.config.js`. We will use it to set the mode of the project, and we need to do this before we get to the options:

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

module.exports = {
  mode: "production", // "production" | "development" | "none"
  entry: "./app/entry", // string | object | array
```

In the preceding code block, the chosen **mode** instructs Webpack to make use of its built-in optimizations accordingly. The **entry** path will default to `./src`. This is where the execution of the application begins and bundling will start.

The following code block will show the rest of the same file:

```
output: {
  path: path.resolve(__dirname, "dist"), // string
  filename: "bundle.js", // string
  publicPath: "/assets/", // string
  library: "MyLibrary", // string,
  libraryTarget: "umd", // universal module definition
  },
```

This section of the code snippets shows options related to how Webpack emits results.

The target directory for all output files must be an absolute path (use the **Node.js** path module).

`filename` indicates the filename template for entry chunks, and `publicPath` refers to the **Uniform Resource Locator** (**URL**) to the output directory, resolved relative to the relevant HTML page. Put simply, this means the file path from the HTML page you may be using to the bundled project files. The remainder of the code refers to the name of the exported library and the nature of the exported library.

The following topic concerns configuration regarding modules. After working on output options, this will be the next logical step in project development:

```
module: {
   rules: [
        {
        test: /\.jsx?$/,
        include: [
          path.resolve(__dirname, "app")
        ],
        exclude: [
          path.resolve(__dirname, "app/demo-files")
        ],
```

The preceding code block includes rules for modules, such as parser options and the configuration of loaders. These are matching conditions, and each accepts a string or a regular expression. The term `test` has the same behavior as `include`. They both must be matched, but this is not the case for `exclude`. `exclude` takes preference over the `test` and `include` options.

For best practice, `RegExp` should only be used in `test` when filenames match. When using arrays of paths, absolute paths should be used in preference to the `include` and `exclude` options. The `include` option should be preferred over the `exclude` method:

```
issuer: { test, include, exclude },
        enforce: "pre",
        enforce: "post",
        loader: "babel-loader",
        options: { presets: ["es2015"] },
},
      {
        test: /\.html$/,
        use: [ "htmllint-loader",
{
```

```
              loader: "html-loader",
              options: {
                / ... /
              }
            }
          ]
        },
```

The preceding code block includes conditions for the issuer, and the origin of the imported elements. The code also includes options to flag the application of these rules, even if they are overridden. This is an advanced option, however.

The reference to `loader` indicates which loader should be applied. This resolves relative to the contextual location. A loader suffix is no longer optional since Webpack 2, for the sake of clarity. There is also space for applying multiple further options and loaders.

In the same configuration, we will explore rules and conditions that can be applied within the same procedure, illustrated in the following code block:

```
{ oneOf: [ / rules / ] },
{ rules: [ / rules / ] },
{ resource: { and: [ / conditions / ] } },
{ resource: { or: [ / conditions / ] } },
{ resource: [ / conditions / ] },
{ resource: { not: / condition / } }],
    /* Advanced module configuration */
  },
  resolve: {
```

The preceding code block includes nested rules, all of which combine with conditions to be useful. By way of explanation, note each of the following commands and what they denote:

- `and` option matches are only made if all conditions are also matched.
- `or` matches apply when a condition is matched—this is the default for arrays.
- `not` indicates if the condition is not matched.

There is also an option for resolving module requests; this does not apply to the resolving of loaders. The following example shows the use of this `resolve` module request:

```
modules: [
      "node_modules",
      path.resolve(__dirname, "app")
    ], extensions: [".js", ".json", ".jsx", ".css"],
    alias: {
            "module": "new-module",
            "only-module$": "new-module",
            "module": path.resolve(__dirname, "app/third/module.js"),
        },
},
  performance: {
    hints: "warning", // enum
    maxAssetSize: 200000, // int (in bytes),
    maxEntrypointSize: 400000, // int (in bytes)
    assetFilter: function(assetFilename) {
    return assetFilename.endsWith('.css') || assetFilename.endsWith('.js');
    }
  },
```

The preceding code block shows the same configuration file we have been following up to now in this section. However, let's take a look at some key elements. Where it states `path.resolve`, this refers to directories in which to look for modules. Directly below this, where it states `], extensions:`, this refers to file extensions that are used.

After this part is code that, in descending order, refers to a list of module name aliases. Modules' aliases are imported relative to the current location context, as illustrated in the following code block:

```
devtool: "source-map", // enum
context: __dirname, // string (absolute path!)
target: "web", // enum
externals: ["react", /^@angular/],
serve: { //object
    port: 1337,
    content: './dist',
    // ...
  },
stats: "errors-only",
```

The `devtool` configuration enhances debugging by adding metadata for the browser. Note that the `source-map` option can be more detailed, but this is at the expense of build speed, and the `web` option indicates the home directory for Webpack. The entry and `module.rules.loader` option is resolved relative to this directory and refers to the environment in which the bundle should run. The `serve` configuration lets you provide options for `webpack-serve` and lets you precisely control which bundle information gets displayed, such as the following:

```
devServer: { proxy: { // proxy URLs to backend development server '/api':
'http://localhost:3000' },
    contentBase: path.join(__dirname, 'public'),
    compress: true,
    historyApiFallback: true,
    hot: true,
    https: false,
    noInfo: true,

  },
  plugins: [

  ],
  // list of additional plugins
```

Let's explain the preceding code block. Where it states `compress: true`, this enables **gzip** compression of contents. The `historyApiFallback: true` part is true for when encountering any 404 page-loading errors. The `hot: true` text refers to having hot module replacement permissible or not; this is subject to `HotModuleReplacementPlugin` being installed first. `https` should be set to `true` for self-signed objects or certificate-authorized objects. If the `noInfo` key is set to `true`, you will only get errors and warnings on hot reloads.

The configuration is done, and we can now run a build. To do this, use the following command:

```
npx webpack-cli init
```

Once the preceding code is run in the command-line environment, the user might be prompted to install `@webpack-cli/init`, if it is not yet installed in the project.

After running `npx webpack-cli init`, more packages may be installed in the project, depending on the choices made during the configuration generation. The following code block shows the printout from running NPX Webpack's CLI initialization:

```
npx webpack-cli init

 INFO For more information and a detailed description of each question,
have a look at https://github.com/webpack/webpack-cli/blob/master/INIT.md
 INFO Alternatively, run `webpack(-cli) --help` for usage info.

 Will your application have multiple bundles? No
 Which module will be the first to enter the application? [default:
./src/index]
 Which folder will your generated bundles be in? [default: dist]:
 Will you be using ES2015? Yes
 Will you use one of the below CSS solutions? No

  babel-plugin-syntax-dynamic-import@6.18.0
  uglifyjs-webpack-plugin@2.0.1
  webpack-cli@3.2.3
  @babel/core@7.2.2
```

```
    babel-loader@8.0.4
    @babel/preset-env@7.1.0
    webpack@4.29.3
    added 124 packages from 39 contributors, updated 4 packages and audited
    25221 packages in 7.463s
    found 0 vulnerabilities


  Congratulations! Your new webpack configuration file has been created!
```

If your output in the CLI looks like the preceding code block, then your configuration has been successful. It's essentially an automated readout from the command line and should signify that all the options set in the previous code block have been recorded.

We have gone through configurations and options, and you should now know the difference and the extent to which each can be used. It's now a natural process to move on to asset management.

Understanding asset management

Assets are primarily managed via the dependency graph, which we covered in `Chapter 1`, *Introduction to Webpack 5.*

Before the great advent of Webpack, developers would use tools such as **grunt** and **gulp** to process these assets and move them from the source folder into the production directory or the development directory (usually named `/build` and `/dist`, respectively).

The same principle was used for JavaScript modules, but Webpack 5 dynamically bundles all dependencies. As every module explicitly states its dependencies, unused modules won't be bundled.

In Webpack 5, any other type of file can now be included, besides JavaScript—for which a loader is used. This method means that all the features possible when using JavaScript can also be utilized.

In the following subsection, we will explore practical asset management. The following topics will be covered:

- Setting up the project for asset management configurations
- Loading **Cascading Style Sheets** (**CSS**) files
- Loading images

- Loading fonts
- Loading data
- Adding global assets

Then, there will be a subsection on wrapping up.

Each subsection will have steps and instructional content to follow. This can become quite a large topic, so hold on tight! We will begin by preparing the configuration of your project.

Setting up the project for asset management configurations

To set things up for asset management configurations in your project, we need to prepare our project index and a configuration file by performing the following steps:

1. Begin by making a minor change to the example project using the `dist/index.html` file, as follows:

```
<!doctype html>
<html>
  <head>
  <title>Asset Management</title>
  </head>
  <body>
   <script src="./bundle.js"></script>
  </body>
</html>
```

2. Now, using `webpack.config.js`, write the following content:

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

module.exports = {
  entry: './src/index.js',
  output: {
   filename: 'bundle.js',
   path: path.resolve(__dirname, 'dist')
  }
};
```

The preceding two code blocks simply show a placeholder index file that we will use to experiment with asset management. The latter code block shows a standard configuration file, with the index file set as the first entry point and names for the output bundle set. This will prepare our project for bundling once we are done with our asset management experiment.

Your project will now be set up for asset management configurations. This guide will now show you how to load CSS files.

Loading CSS files

The example project will now show the inclusion of CSS. This is a very easy thing to get to grips with, as most frontend developers beginning with Webpack 5 should know it well.

To load CSS and run a build, perform the following steps:

1. Firstly, install and add `style-loader` and `css-loader` to the project's module configuration, using the following command-line instruction:

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

2. Next, make the following additions to the `webpack.config.js` file:

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

module.exports = {
  entry: './src/index.js',
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist')
  },
 module: {
   rules: [
     {
       test: /\.css$/,
       use: [
         'style-loader',
         'css-loader'
       ]
     }
   ]
 }
};
```

As you can see from the previous code block, the following additions refer to the use of `style-loader` and `css-loader` toward the end of the block. So that you don't get any errors, you should ensure that your code mirrors the example.

> The difference between `style-loader` and `css-loader` is that the former determines how styles will be injected into a document—such as with style tags, whereas the latter will interpret `@import` and `require` statements, and then resolve them.

It is recommended that both loaders are used together, as almost all **CSS** operations involve a combination of these methods at some point in the project's development.

In Webpack, regular expressions are used to determine which files should be looked for and be served to a specific loader. This permits the import of a style sheet into the file that depends on it for styling. When that module is run, a `<style>` tag with the stringified CSS will be inserted into `<head>` of the HTML file.

3. Now, navigate to the directory structure, which we can see in the following example:

```
webpack5-demo
package.json
webpack.config.js
/dist
bundle.js
index.html
/src
style.css
index.js
/node_modules
```

We see from this structure that there is a style sheet named `style.css`. We are going to use this to demonstrate the use of `style-loader`.

4. Enter the following code in `src/style.css`:

```
.hello {
  color: blue;
}
```

This preceding code simply creates a color class style that we will use to attach a style to our frontend and show how the CSS load works.

5. Likewise, make the following append to `src/index.js`:

```
import _ from 'lodash';
import './style.css';

function component() {
  const element = document.createElement('div');

  // Lodash, now imported by this script
  element.innerHTML = _.join(['Hello', 'Webpack'], ' ');
  element.classList.add('hello');

  return element;
}

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

   The preceding code all takes place inside the `index.js` file. It essentially creates a JavaScript function that appends a `<div>` element inside whichever files call it from the browser. In this example, it will be the `index.html` file, aforementioned in the directory structure illustration. The preceding code will then "join" an **HTML** element to the web page with text stating, '`Hello, Webpack`'. We will use this to test whether `style-loader` and `css-loader` have been used correctly. As the commented part of the script states, this element appendment will automatically import `lodash` for use with Webpack.

6. Finally, run the `build` command, as follows:

```
npm run build

...
    Asset       Size  Chunks             Chunk Names
bundle.js   76.4 KiB       0  [emitted]  main
Entrypoint main = bundle.js
...
```

When the `index.html` file is opened in a browser window, you should see that '`Hello Webpack`' is now styled in blue.

To see what happened, inspect the page (not the page source, as it won't show the result) and look at the page's head tags. This is best done using Google's Chrome browser. It should contain the style block that we imported in `index.js`.

You can—and in most cases, should—minimize CSS for better load times in production.

The next natural step is to work on adding images. Images can be added to your project the same way as for any website application. Place these images in whatever desired format in an image folder. This must be in the `/src` folder, but they can be located anywhere in there. The next procedure is the loading of images with Webpack, and we will go through this now.

Loading images

Now, let's try loading images and icons using the file loader, which can be easily incorporated into our system.

To do this, perform the following steps:

1. Using the command line, install `file-loader`, as follows:

```
npm install --save-dev file-loader
```

2. Now, using the usual `webpack.config.js` Webpack configuration file, make the following amendments to it:

```
const path = require('path');
module.exports = {
  entry: './src/index.js',
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist')
  },
  module: {
    rules: [
      {
        test: /\.css$/,
        use: [
          'style-loader',
          'css-loader'
        ]
      },
      {
        test: /\.(png|svg|jpg|gif)$/,
        use: [
           'file-loader'
```

```
                ]
              }
            ]
          }
        };
```

Now, because of the code in the previous block, when you import an image, that image will be processed to the output directory, and the variable associated with that image will contain the final **URL** of that image after processing. When using the `css-loader`, a similar process will occur for the **URL** of the image file within your **CSS** file. The loader will recognize that this is a local file and replace the local path with the final path to the image in your output directory. The `html-loader` handles `<img src="./my-image.png" />` in the same manner.

3. Next, to start adding an image, you need to navigate to the project file structure, which looks like this:

```
webpack5-demo
|- package.json
|- webpack.config.js
|- /dist
  |- bundle.js
  |- index.html
|- /src
  |- icon.png
  |- style.css
  |- index.js
|- /node_modules
```

This structure seems very similar to the immediately previous project directly used for the *Loading CSS files* tutorial for the most part, except for the addition of the `icon.png` image file.

4. Then, navigate to the JavaScript frontend file, `src/index.js`. The following code block shows the content:

```
import _ from 'lodash'; import './style.css';
import Icon from './icon.png';
function component() {
    const element = document.createElement('div');
    // Lodash, now imported by this script
        element.innerHTML = _.join(['Hello', 'Webpack'], ' ');
        element.classList.add('hello');
    // Add the image to our existing div.
    const myIcon = new Image();
```

```
        myIcon.src = Icon;
        element.appendChild(myIcon);
        return element;
    }
    document.body.appendChild(component());
```

It can be seen from this preceding block that the import of **lodash** will allow the **HTML** of your page to be appended with the `Hello Webpack` text. Beyond that, this code simply sets up our web page with our image, using some crafty JavaScript. It first creates a variable called `Icon` and gives it the value of the image file's **URL**. Later in the code, it then assigns this to the source of an element called `myIcon`.

5. From here, we want to set some very basic styles to handle our image with the style sheet. In the `src/style.css` file, append the following code:

```
.hello {
  color: red;
  background: url('./icon.png');
}
```

It will, of course, show your image icon as the background for `div` we assigned code to in the **HTML**, with the text turned **red** wherever the `.hello` class is applied.

6. Run a new build and open up the `index.html` file, as follows:

```
npm run build

...
Asset                                  Size          Chunks
Chunk Names
da4574bb234ddc4bb47cbe1ca4b20303.png  3.01 MiB          [emitted]
[big]
bundle.js                              76.7 KiB       0  [emitted]
main
Entrypoint main = bundle.js
...
```

This will create the effect of the icon repeating as a background image. There will also be an `img` element beside the `Hello Webpack` text.

Often, this command can go wrong, even for experienced developers. For example, the image might not load at all, be too large, or will not be bundled correctly. This can be caused by a combination of factors, including the use of the loader in an unusual way. Webpack may also experience code skipping when using long filenames.

If this is the case, simply repeat the steps, as follows:

1. Install `file-loader` using the command line.
2. Alter the `webpack.config.js` file, as described in the preceding example.
3. Check that the project file structure and index file are formatted correctly to load the image file.
4. Check that the **CSS** is also formatted the way you want it to be.
5. Then, run the build using `npm` and the command line.
6. Check that the index file is loading the image correctly.

If the element is inspected, the actual filename can be seen to have changed to something similar to `da4574bb234ddc4bb47cbe1ca4b20303.png`. This means that Webpack found our file in the source folder and processed it.

That gives you a solid framework for managing images. In the next subsection, we will discuss the management of fonts as Webpack assets.

Loading fonts

Now, we will examine fonts in the context of assets. The file and URL loaders will take any file you load through them and output it to your build directory. This means we can use them for any kind of file, including fonts.

We will begin by updating the Webpack configuration JavaScript file, which is needed to handle fonts, as follows:

1. Ensure the update of the configuration file is made. We are updating our usual `webpack.config.js` configuration file here, but you will notice toward the end that some font types, such as `.woff`, `.woff2`, `.eot`, `.ttf`, and `.otf`, have been added, as illustrated in the following code block:

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

module.exports = {
  entry: './src/index.js',
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist')
  },
  module: {
    rules: [
      {
        test: /\.css$/,
        use: [
          'style-loader',
          'css-loader'
        ]
      },
      {
        test: /\.(png|svg|jpg|gif)$/,
        use: [
          'file-loader'
        ]
      },
     {
       test: /\.(woff|woff2|eot|ttf|otf)$/,
       use: [
         'file-loader'
       ]
     }
    ]
  }
};
```

This configuration allows Webpack's `file-loader` to incorporate the font type, but we still have to add some font files to our project.

2. We can now perform the essential task of adding the font to the source directory. The following code block illustrates a file structure, indicating where the new font files can be added:

```
webpack5-demo
|- package.json
|- webpack.config.js
|- /dist
  |- bundle.js
  |- index.html
|- /src
  |- sample-font.woff
  |- sample-font.woff2
  |- icon.png
  |- style.css
  |- index.js
|- /node_modules
```

Note the `src` directory and the `sample-font.woff` and `sample-font.woff2` files. These two files should be replaced with any font files you choose. The **Web Open Font** (**WOFF**) format is generally recommended for use with Webpack projects.

Fonts can be incorporated in the styling of your project by using the `@font-face` declaration. The local URL directive will be found by Webpack the same way it handles images.

3. Update the style sheet using the `src/style.css` file to include the sample font on our home page. This is done with the use of a font declaration at the top of the code block and a class definition below that, as shown in the following code block:

```
@font-face {
  font-family: 'SampleFont';
  src:  url('./sample-font.woff2') format('woff2'),
        url('./sample-font.woff') format('woff');
  font-weight: 600;
  font-style: normal;
}

.hello {
  color: blue;
  font-family: 'SampleFont';
  background: url('./icon.png');
}
```

Note that you must change the `'SampleFont'` text to one corresponding to your chosen font file. The previous code shows the loading of the font via CSS and the setting of custom values such as `font-weight` and `font-style`. The **CSS** code then uses the `.hello` class to assign that font to any prospective **HTML** element. Note that we have already prepared our `index.html` file for this in the two previous tutorials, *Loading CSS files* and *Loading images*.

4. Now, run a `npm` build in development mode using the command-line utility as per usual, like this:

```
npm run build

...
                                 Asset      Size  Chunks
Chunk Names
5439466351d432b73fdb518c6ae9654a.woff2  19.5 KiB          [emitted]
 387c65cc923ad19790469cfb5b7cb583.woff  23.4 KiB          [emitted]
  da4574bb234ddc4bb47cbe1ca4b20303.png  3.01 MiB          [emitted]
[big]
bundle.js                                 77 KiB       0  [emitted]
main
Entrypoint main = bundle.js
...
```

Open up `index.html` again and see whether the `Hello Webpack` sample text we are using has changed to the new font. If all is well, you should see the changes.

That should serve as a simple tutorial to understand font management. The next section will cover data management of files such as **Extensible Markup Language** (**XML**) and **JavaScript Object Notation** (**JSON**) files.

Loading data

Another useful asset that can be loaded is data. Data is an incredibly important asset to be loaded. This will include files such as **JSON**, **Comma-Separated Values** (**CSV**), **Tab-Separated Values** (**TSV**), and **XML** files. Using a command such as `import Data from './data.json'` will work by default, meaning **JSON** support is built into Webpack 5.

To import the other formats, a **loader** must be used. The following subsection demonstrates a method for handling all three. The following steps should be taken:

1. To begin, you must install the `csv-loader` and `xml-loader` loaders using the command line, as follows:

   ```
   npm install --save-dev csv-loader xml-loader
   ```

   The preceding code block simply shows the command line to install two data loaders.

2. Open and append the `webpack.config.js` configuration file, and ensure that it looks like the following example:

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

   module.exports = {
     entry: './src/index.js',
     output: {
       filename: 'bundle.js',
       path: path.resolve(__dirname, 'dist')
     },
     module: {
       rules: [
         {
           test: /\.css$/,
           use: [
             'style-loader',
             'css-loader'
           ]
   ```

```
          },
          {
            test: /\.(png|svg|jpg|gif)$/,
            use: [
              'file-loader'
            ]
          },
          {
            test: /\.(woff|woff2|eot|ttf|otf)$/,
            use: [
              'file-loader'
            ]
          },
          {
            test: /\.(csv|tsv)$/,
            use: [
              'csv-loader'
            ]
          },
          {
            test: /\.xml$/,
            use: [
              'xml-loader'
            ]
          }
        ]
      }
    };
```

In the preceding code block, the lower portion shows the use of `csv-loader` and `xml-loader`. It is this amendment that will be needed this time to load the data into our project.

3. Next, we must add a data file to the source directory. We will be adding an **XML** data file to our project, shown in bold text in the following code block:

```
webpack5-demo
|- package.json
|- webpack.config.js
|- /dist
  |- bundle.js
  |- index.html
|- /src
  |- data.xml
  |- samplefont.woff
  |- sample-font.woff2
  |- icon.png
  |- style.css
```

```
    |- index.js
  |- /node_modules
```

Take a look at the preceding `data.xml` file in the `src` directory of your **project** folders. Let's take a closer look inside this file to see what the data is, as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<note>
  <to>Tim</to>
  <from>Jakob</from>
  <heading>Reminder</heading>
  <body>Call me tomorrow</body>
</note>
```

As you can see from the previous block of code, the contents are a very basic **XML** dataset. We are going to use this to import the **XML** data into our project's `index.html` page, and we will need this to be formatted correctly to ensure that it works.

Any one of those four types of data (**JSON**, **CSV**, **TSV,** and **XML**) can be imported, and the data variable you import it to will contain parsed JSON.

4. Be sure to amend the `src/index.js` file to expose the data file. Note the import of `./data.xml`, as illustrated in the following code block:

```
import _ from 'lodash';
import './style.css';
import Icon from './icon.png';
import Data from './data.xml';

function component() {
  const element = document.createElement('div');

  // Lodash, now imported by this script
  element.innerHTML = _.join(['Hello', 'Webpack'], ' ');
  element.classList.add('hello');

  // Add the image to our existing div.
  const myIcon = new Image();
  myIcon.src = Icon;

  element.appendChild(myIcon);

  console.log(Data);

  return element;
```

```
        }

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

The addition of the `import` function, and little else, is all we need this time to demonstrate the usage. Anyone familiar enough with JavaScript will also know how to make use of this to run their particular project very easily.

5. Run a build and check that the data loads correctly, as follows:

```
npm run build
```

Once a `npm` build is run, the `index.html` file can be opened. Examining the console (such as under **developer tools** when using Chrome) will show the data being logged after import.

Something related, but more to do with project architecture, is the arrangement of global assets for project consumption. Let's explore that in the following subsection.

Adding global assets

Loading assets in the aforementioned ways allows modules to be grouped together in a more intuitive, practical, and usable way.

Instead of using a global assets directory containing every asset, assets can be grouped with the code that utilizes them. The following filing structure or tree demonstrates a very practical and usable example:

```
|- /assets
|- /components
|   |- /my-component
|   |   |- index.jsx
|   |   |- index.css
|   |   |- icon.svg
|   |   |- img.png
```

The preceding example makes your code a lot more portable. If you want to place one component in another directory, simply copy or move it there. Alternatively, a base directory could be used if your development works along old-fashioned lines. Also, aliasing is an option.

Wrapping up the tutorial with best practice

It's been a long tutorial and some of your code might have gone astray. It's good practice to clean up this code and check for anything erroneous.

Cleaning up is a good habit to get into. We won't be using a lot of assets in the next section, *Understanding output management*, so let's start there.

1. We begin wrapping up with the project directories, **project tree**. Let's check them to see whether they are right. It should something like the following:

```
webpack5-demo
|- package.json
|- webpack.config.js
|- /dist
  |- bundle.js
  |- index.html
|- /src
  |- data.xml
  |- sample-font.woff
  |- sample-font.woff2
  |- icon.png
  |- style.css
  |- index.js
|- /node_modules
```

As we are wrapping up, you should remove the files that correspond to the emboldened text in the preceding code block.

This should give you a good idea of what your project files and folders look like. Ensure that all the files we have been using are there and in the appropriate folder.

2. Let's check the formatting of our configuration.

A lot of work has been done on `webpack.config.js`, and we must take care that the contents are formatted correctly. Please refer to the following code block and check it against your own to ensure this is correct. It is often useful to count the number of `{` and beautify your code with a conventional structure to make this process easier:

```
const path = require('path'); module.exports = {
  entry: './src/index.js',
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist')
  },
  module: {
    rules: [
      {
        test: /\.css$/,
        use: [
          'style-loader',
          'css-loader'
        ]
      },
      {
        test: /\.(png|svg|jpg|gif)$/,
        use: [
          'file-loader'
        ]
      },
      {
        test: /\.(woff|woff2|eot|ttf|otf)$/,
        use: [
          'file-loader'
        ]
      },
      {
        test: /\.(csv|tsv)$/,
        use: [
          'csv-loader'
        ]
      },
      {
        test: /\.xml$/,
        use: [
          'xml-loader'
        ]
      }
    ]
  }
};
```

Notice the extensive reference to CSS, images files, fonts such as `.woff`, and data files in separate handlers such as `.csv` and `.xml`. All of this is important, and you should take the time to make sure the scripting is accurate as this has been an extensive topic and practical exercise, so a lot of things could have been overlooked.

3. Next, we need to check the scripting of the `src/index.js` file, as follows:

```
import _ from 'lodash';
import './style.css';
import Icon from './icon.png';
import Data from './data.xml';

function component() {
  const element = document.createElement('div');

  // Lodash, now imported by this script
  element.innerHTML = _.join(['Hello', 'Webpack'], ' ');
  element.classList.add('hello');

  // Add the image to our existing div.
  const myIcon = new Image();
  myIcon.src = Icon;

  element.appendChild(sampleIcon);

  console.log(Data);

  return element;
}

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

   Once again, we are wrapping up here so that the code is reusable after having followed multiple tutorials using it, so be sure to remove the emboldened text in your version.

We've gone through an extensive list of asset management operations and have concluded with the project tidying process. All of your code should look like the previous code blocks in the wrapping-up section for it to operate correctly.

You should now have a sound understanding of how Webpack manages these assets, and how to manage them when working with Webpack. With the file structure and code cleaned through and tidied, we are now best placed to begin output management.

Understanding output management

The output refers to the bundles created from the source files. The source files are referred to as the input in Webpack. Output management refers to the management of these newly bundled files. Depending on the mode Webpack was running in when the build began, these bundles will either be development or production bundles.

The process by which Webpack produces the output or bundle from the source files is called compiling. Compiling is the process by which Webpack 5 assembles the information, including assets, files, and folders. The topic of configuration relates to the various options and configurations possible in Webpack, which will alter the style and method of compilation.

Development bundles permit some customization (such as local testing), but production bundles are the finished and fully compressed versions, ready for publication.

During this chapter, assets have been manually added to the index **HTML** file. As the project grows, manual handling will be difficult, especially when using multiple bundles. That being said, a few plugins exist that make this process much easier.

We will now discuss those options, but begin with preparing your now very busy project structure, which will become an increasingly important practice as the project develops.

Output management tutorial preparation

First, let's adjust our project file structure tree a little and make things easier. This process follows these next steps:

1. Begin by locating the `print.js` file in the project folder, as follows:

```
webpack5-demo
|- package.json
|- webpack.config.js
|- /dist
|- /src
  |- index.js
  |- print.js
|- /node_modules
```

Note the addition to our project structure—the `print.js` file, specifically.

2. Append the code by adding some logic to the `src/print.js` file, as follows:

   ```
   export default function printIt() {
     console.log('This is called from print.js!');
   }
   ```

   You should use the `printIt()` JavaScript function, as seen in the preceding code block, in the `src/index.js` file.

3. Prepare the `src/index.js` file to import the required external files and write a simple function in it to allow interaction, as follows:

   ```
   import _ from 'lodash';
   import printMe from './print.js';

   function component() {
     const element = document.createElement('div');
     const btn = document.createElement('button');

     element.innerHTML = _.join(['Hello', 'Webpack'], ' ');

     btn.innerHTML = 'Click here then check the console!';
     btn.onclick = printIt();

     element.appendChild(btn);

     return element;
   }

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

   We've updated our `index.js` file with an import of the `print.js` file at the top and a new `printIt();` function button at the bottom.

4. We must update the `dist/index.html` file. This update is done in preparation for the entries to be split out, and is illustrated in the following code block:

   ```
   <!doctype html>
   <html>
     <head>
       <title>Output Management</title>
       <script src="./print.bundle.js"></script>
     </head>
     <body>
       <script src="./app.bundle.js"></script>
     </body>
   </html>
   ```

This preceding **HTML** script will load in the `print.bundle.js` file and, below that, the `bundle.js` and `app.bundle.js` files.

5. Next, ensure that the configuration of the project is in line with dynamic entry points. The `src/print.js` file will be added as a new entry point. The outputs will also be changed so that bundles' names will be dynamically generated based on entry point names. In `webpack.config.js`, there is no need to change the directory names due to this automatic process. The following code block shows the content of `webpack.config.js`:

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

module.exports = {
  entry: './src/index.js',
  entry: {
    app: './src/index.js',
    print: './src/print.js'
  },
  output: {
    filename: 'bundle.js',
    filename: '[name].bundle.js',
    path: path.resolve(__dirname, 'dist')
  }
};
```

   The configuration simply set up new entry points for the new files we have been working on, `index.js` and `print.js`.

6. Make sure you perform a build. Once you run a `npm` build, you will see the following:

```
...
Asset             Size      Chunks                      Chunk Names
app.bundle.js     545 kB    0, 1  [emitted]    [big]   app
print.bundle.js    2.74 kB  1     [emitted]            print
...
```

After opening the `index.html` file in your browser, you will see that Webpack generated the `print.bundle.js` and `app.bundle.js` files. We should now check that it worked! If the entry point names are changed or new ones added, the **index HTML** would still reference the older names. This can be corrected with `HtmlWebpackPlugin`.

Setting up the HtmlWebpackPlugin

The `HtmlWebpackPlugin` will allow Webpack to process HTML files that contain JavaScript, for instance. To start working with it, we need to install it using the command line, and then set the configuration correctly, as follows:

1. First, install the plugin using the command-line utility, and then adjust the `webpack.config.js` file, as follows:

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

The preceding code block shows the installation of the `HtmlWebpackPlugin` for use in our project.

2. Next, we have to incorporate the plugin into our configuration. Let's take a look at the `webpack.config.js` file when associated with this plugin, as follows:

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

module.exports = {
  entry: {
    app: './src/index.js',
    print: './src/print.js'
  },
  plugins: [
    new HtmlWebpackPlugin({
      title: 'Output Management'
    })
  ],
  output: {
    filename: '[name].bundle.js',
    path: path.resolve(__dirname, 'dist')
  }
};
```

Note the use of the `require` expression and the `plugins:` option key, both of which serve to permit the use of the plugin.

Before a build is run, note that the `HtmlWebpackPlugin` will generate its `index.html` file by default, even though there already is one in the `dist/` folder. As a result, the existing file will be overwritten.

For best practice, make a copy of the existing index file and name it something like `index2.html`. Place this new file next to the original, and then run the build.

3. Now, run the build using your command-line utility. Once this is done, you will see the following result in the command-line utility window, indicating a successful bundling:

```
...
           Asset       Size  Chunks                         Chunk Names
 print.bundle.js     544 kB       0  [emitted]  [big]  print
   app.bundle.js    2.81 kB       1  [emitted]         app
      index.html  249 bytes          [emitted]
...
```

Opening the `index.html` file in your code editor or **Notepad** will reveal that the plugin has created a new file, and all the bundles are automatically added.

Also, why not look at `html-webpack-template`, which provides a few extra features on top of the default template?

That concludes our tutorial of Webpack's `HtmlWebpackPlugin`. In the following subsection, we will again embark on some tidying up in your project directory.

Cleaning up the distribution directory

During this project development, the `/dist` folder will become quite cluttered. Good practice involves good organization, and this involves cleaning the `/dist` folder before each build. There is a `clean-webpack-plugin` plugin that can be used to do this for you, as follows:

1. Start by installing the `clean-webpack-plugin`. The following example shows you how to do this:

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

Once the plugin is installed, we can delve back into the configuration file.

2. Using `webpack.config.js`, make the following entry in the file:

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

module.exports = {
  entry: {
    app: './src/index.js',
    print: './src/print.js'
  },
  plugins: [
  new CleanWebpackPlugin(),
    new HtmlWebpackPlugin({
      title: 'Output Management'
    })
  ],
  output: {
    filename: '[name].bundle.js',
    path: path.resolve(__dirname, 'dist')
  }
};
```

   Note the use of `CleanWebpackPlugin`, proceeding with the `const` qualifier. This will be the addition of the `module.export` plugin option, which creates a new function associated with the plugin and will make the plugin usable by Webpack during compilation.

3. You should now run an `npm` build, which will output a bundle to the`/dist` distribution folder.

   After you run a `npm` build, the `/dist` folder can be inspected. You should only see newly generated files and no more old ones, assuming the process behaved correctly.

We've been generating a lot of files, and to help us keep track, there is something called the manifest, which we will cover next.

Utilizing the manifest

Webpack can know which files are being generated thanks to the manifest. This allows the software to track all output bundles and to map the modules. To manage outputs in other ways, it would be a good idea to utilize the manifest.

Webpack essentially categorizes code by three types: source code written by the developer; vendor code written by a third party; and Webpack's runtime manifest, which conducts the interactions of all the modules.

The runtime and manifest data are what Webpack needs to connect your modular application while it runs in the browser.

If you decide to improve performance by using browser caching, this process will become an important thing to get to grips with.

By using content hashes within your bundle filenames, you can indicate to the browser when a file's contents have changed, and are thereby invalidating the cache. This is caused by the injection of the runtime and manifest, which changes with every build.

Webpack has `WebpackManifestPlugin` that can extract manifest data into a **JSON** file.

Now that you've learned about dynamically adding bundles to your HTML, let's dive into the development guide. Or, if you want to dig into more advanced topics, we would recommend rereading the *Code splitting* section of this guide in the previous chapter, `Chapter 2`, *Working with Modules and Code Splitting*.

Exploring Webpack 5 options

Options are a set of variables that can be altered using the CLI. Configuration, on the other hand, is done by altering the file contents. Options' settings can be adjusted using the configuration file, however. The following is a list of the options supported by Webpack 5 currently:

- **Asynchronous Module Definition** (**AMD**)
- Bail
- Cache
- Loader
- Parallelism
- Profile
- Records Path
- Records Input Path
- Records Output Path
- Name

The following sections describe and illustrate each option is a bit more detail. We will begin with something that, after a cursory examination of Webpack's configuration, might find you scratching your head: the AMD option.

AMD

AMD is an `object bool: false` option. It is also an acronym for Asynchronous Module Definition. Essentially, it is a format to provide a solution for modular JavaScript that developers. The format itself is a proposal for defining modules in which both the module and dependencies can be asynchronously loaded.

This allows you to set the value of `require.amd` or `define.amd`. Setting `amd` to `false` will disable **AMD** support.

Look inside the `webpack.config.js` file, as follows:

```
module.exports = {
  amd: {
    jQuery: true
  }
};
```

Popular modules for **AMD**, such as jQuery versions 1.7.0 to 1.9.1, will only register as an **AMD** module if the loader indicates that special allowances are permitted for multiple versions being used on one page. Another similar option, in terms of being a Boolean variable, is **Bail**. Let's take a closer look.

Bail

Bail is a `bool` value. This will force Webpack to exit its bundling process. It will cause Webpack to fail out on the first error instead of tolerating it. By default, Webpack will log these errors in red text in the Terminal (the browser console also, when using **HMR**) but will continue bundling.

To enable this option, open `webpack.config.js`, as follows:

```
module.exports = {
  bail: true
};
```

This will be very helpful if you want Webpack to exit a bundling process in certain circumstances. Perhaps you only want part of a project bundled. It's really up to you. Next up is caching.

Cache

A cache is a term that refers to a `bool` object. It will cache the generated Webpack modules and chunks; this improves the speed of the build. It does this by keeping a reference to this object between compiler calls that can be shared. Caching is enabled by default while in **watch mode** and development mode.

In watch mode, after the initial build, Webpack will continue to watch for changes in any of the processed files. Essentially, the **Webpack configuration JavaScript** file should include the `watch: true` operand inside the `module.export` operator.

To enable caching, manually set it to `true` using webpack.config.js, as in the following example:

```
module.exports = {
  cache: false
};
```

The `webpack.config.js` file shows the configuration needed to allow shared caching, as follows:

```
let SharedCache = {};

module.exports = {
  cache: SharedCache
};
```

The two previous examples show the caching configuration set to `false` and `sharedCache`. These are the two Boolean values that can be set in Webpack.

Warning: The cache should NOT be shared between calls with different options.

There are a few more options that can be set in Webpack: Loader, Parallelism, Profile, Records Path, Records Input Path, Records Output Path, and Name. Let's go through each one, one at a time, right now.

Loader

This is expressed as `loader` and exposes custom values in the loader context, as illustrated in the following code block:

```
use: [
    {
       loader: worker-loader
    }
]
```

You can see from the preceding code example how this option can be used in the configuration file. This example should look familiar to anyone following this guide and configuring loaders. This example uses `worker-loader` by way of example only. Some of these options are a Boolean or a binary, such as the `profile` option, described next.

Profile

The `profile` option will capture a profile of the application, which can then be dissected using the **Analyze** tool, as illustrated in the following code snippet:

```
profile: true,
```

Note that this is a Boolean value. You can use the `StatsPlugin` for more control over the profile. This can also be combined with the `parallelism` option for better results.

Parallelism

Parallelism will limit the number of parallel processed modules. This can be used to fine-tune performance or for more reliable profiling. The following example gives the limit number as `1`, but you can alter this as you wish:

```
parallelism: 1
```

Webpack 5 permits the use of modules being processed in parallel as well as bundling in parallel. This can eat up memory, so this option should be noted on larger projects.

As your project gets more complex, you may want to record the compilation process, which can help with—among other things—tracing bugs and errors. **Records Path** will help you do this, and we will take a closer look at that now.

Records Path

The Records Path option is expressed as a string. This option should be used to generate a **JSON** file containing records. These are pieces of data used to store module identifiers across numerous builds. This can be used to track how modules alter between builds. To generate one, simply specify a location, as in the following example, using the `webpack.config.js` file:

```
module.exports = {
  recordsPath: path.join(__dirname, 'records.json')
};
```

Records are useful if you have a complex project that uses code splitting. This recorded data can be utilized to ensure that caching is behaving correctly when working with split bundles.

Even though the compiler generates this file, the source control should be used to track it and keep a history of its usage over time.

Setting `recordsPath` will also set `recordsInputPath` and `recordsOutputPath` to the same location.

Records Input Path

This option is expressed as a string, `recordsInputPath`, as illustrated in the following code block:

```
module.exports = {
recordsInputPath: path.join(__dirname, 'records.json'),
};
```

It will specify the file from which the last set of records is read, and can be used to rename a records file. Related is the Records Output Path option, and we will go over that now.

Records Output Path

Records Output Path is a **string** that specifies where the records should be written. The following code example shows how you might use this option in combination with `recordsInputPath` when renaming a records file. We will use `webpack.config.js` to do this:

```
module.exports = {
  recordsInputPath: path.join(__dirname, 'records.json'),
  recordsOutputPath: path.join(__dirname, 'outRecords.json')
};
```

The preceding code will set the location where records are written. If it's an input record, it would be written to `__dirname/records.json`. If it's an output record, it will be written to `__dirname/newRecords.json`.

The next option we need to discuss is the Name option.

Name

The **Name** option is expressed as a **string** and denotes the name of the configuration. It should be used when loading multiple configurations. The following example shows code that should form part of the `webpack.config.js` file:

```
module.exports = {
  name: 'admin-app'
};
```

The preceding code is useful when using multiple configuration files. The code will name this configuration file as `admin-app`. This gives you a long rundown of options and how to use them. Let's now take a look at what we have covered throughout this chapter.

Summary

This chapter has followed the practices of configuration files, asset management, and options. This chapter began by taking you, the reader, through the various capabilities of Webpack and configuration, and exploring how to manage those assets and control content accordingly. You were guided through both input and output management, and the loading of external content such as fonts and images. From there, this chapter took us through options and the difference between the two, explaining to the reader what can be achieved using options that can be set up simply through configuration.

You were then guided through common option methods and how to use them. You are now fully versed in options and configurations. You should now know the difference between the two and the best methods to adopt, given any number of possibilities that would require either technique.

In the next chapter, we will pry into the world of APIs' loaders and plugins. These features of Webpack expound the capabilities of the platform, springboarding from configurations, and options.

You will learn the difference between loaders and plugins, and the essential nature of loaders to use languages and scripts that are not supported by default. Much of these loaders are supplied by third-party developers, so plugins fill gaps in capability where a loader cannot be used—and vice versa.

The similar topic of APIs will then be expanded upon. APIs are essentially used to connect an application to remote applications on a network. This gives them a similar characteristic to loaders, and they are often used where a native script is not available.

Questions

To assist in your learning, here are a set of questions on the topics covered in this chapter (you will find the answers in the back matter of this guide):

1. What is the difference in Webpack 5 between configurations and options?
2. What is a config flag?
3. Which loader is required for loading images into Webpack projects?
4. What type of data file does Webpack allow the import of without the use of a loader?
5. What does Webpack's manifest record indicate?
6. What does the Bail option do?
7. What does the Parallelism option do?
8. What does the Records Input Path option do?
9. What will setting AMD to `false` do?
10. What is compiling?

4
APIs, Plugins, and Loaders

An **Application Programming Interface** (**API**) is typically used for interfaces between remotely stationed programs, such as when a company offers partial access to their website's functionality as part of an integrated system through, say, a mobile app.

Webpack seeks to compile and optimize localized code, so knowing the difference between localized code and an external API is essential for operating the software.

Plugins and loaders are similar. A loader essentially instructs Webpack on how to handle specific tasks involved in more unusual programming languages and bundling. Loaders are often developed by the user community, rather than by in-house Webpack developers. Plugins, on the other hand, offer some of the processes that loaders don't currently offer and so are more general in their operation than loaders. A concise explanation of each feature with detailed examples will follow during the course of this chapter.

Webpack 5 offers developers a rich plugin interface. Most of the features within Webpack use this plugin interface, making Webpack quite flexible to use.

This chapter will explore plugins, loaders, and APIs, as well as the salience of each and how each feature plays its part in the operation of Webpack.

The topics discussed in this chapter are as follows:

- Loaders
- APIs
- Plugins

Loaders

Loaders are fundamental to Webpack and many of them enable greater functionality, particularly with scripts and frameworks that are not native ECMAScripts, such as JavaScript and JSON.

This chapter aims to give you a broad overview of the available loaders as well as some that you may need to buy. When working with a salient or unique code that is specific to your project, you should search the Webpack online registry to ensure that the code can be transpiled.

In particular, this section will discuss the following loaders:

- `cache-loader`
- `coffee-loader`
- `coffee-redux-loader`
- `worker-loader`
- `cover.js`
- `i18n-loader`
- `imports-loader`
- `polymer-webpack-loader`
- `script-loader`
- `source-map-loader`
- `less-loader`

We'll discuss and give examples of each loader, where appropriate, although some may need no real elaboration. Let's begin with `cache-loader`.

cache-loader

Caching is something that we alluded to in the previous chapter. `cache-loader` allows caches to be made from a loader. We can set it up as follows:

1. Begin by installing the loader through the **Command-Line Interface** (**CLI**), as follows:

```
npm install --save-dev cache-loader
```

Note that other loaders are executed in our configuration (see the following code) and a cache will be made of any of the resulting outputs made by the loader that start with `cache-loader`. This will be made, by default, in the project folder but can also be configured to be made in the cache in a database.

2. To configure this, use `webpack.config.js`:

```
module.exports = {
 module: {
  rules: [
  {
    test: /\.ext$/,
    use: ['cache-loader', 'babel-loader'],
    include: path.resolve('src'),
   },
  ],
 },
};
```

Note that where `cache-loader` is placed in the configuration is where it should always be placed, ahead of other loaders.

That explains the installation and use of `cache-loader`. We'll go through a few other loaders in the same way, starting with `worker-loader`. As it will need to be declared first, we should discuss it first, even though it is always used in a chain or sequence with other loaders.

worker-loader

`worker-loader` essentially gives the developer a solution for getting the program to handle large computation tasks in the background. To get the loader up and running, we will take the following steps:

1. To begin with, use the command line to install `worker-loader`:

```
npm install worker-loader --save-dev
```

Alternatively, there is an inline way to import `worker-loader` from the `App.js` file. In this file, which is present in any Webpack project directory, make the following amendment, if it hasn't been done already:

```
import Worker from 'worker-loader!./Worker.js';
```

2. Once you have imported the loader, perform a configuration using `webpack.config.js`:

```
{
 module: {
  rules: [
  {
   test: /\.worker\.js$/,
   use: {
      loader: 'worker-loader'
      }
    }
   ]
  }
 }
```

   `use` refers to the configuration needed to allow access to the loader.

3. Write the following code in `App.js` to allow this file to be an export location for the loader:

```
import Worker from './file.worker.js';
const worker = new Worker();
worker.postMessage({ a: 1 });
worker.onmessage = function (event) {};
worker.addEventListener("message", function (event) {});
```

   The preceding code also adds an `event` listener to allow testing later in the development console.

4. Finally, run Webpack via your preferred method to see the results:

```
npm run build
```

You should see `worker-loader` installed and imported from `App.js` if you chose that method. This can be observed in the console window or by viewing the page source.

This gives you a couple of choices for utilizing `worker-loader`, either through the command-line utility or through the configuration of the `App.js` file. Next, we'll discuss `coffee-loader`.

coffee-loader

CoffeeScript is a simplified form of JavaScript, but it isn't exactly JavaScript, so a loader must be used to work with it in Webpack.

Let's go through the following steps to be able to use `coffee-loader`:

1. Begin by installing `coffee-loader`. To install the loader, use the following command line:

   ```
   npm install --save-dev coffee-loader
   ```

2. Ensure that you are using the recommended configuration of the loader for the sake of testing and to use the `literate` key. To allow testing, load `coffee-loader` and set the `literate` key to `true`. The `literate` key will ensure that the loader's usage is interpreted by the compiler :

   ```
   module.exports = {
       module: {
         rules: [
          {
             test: /\.coffee.md$/,
             use: [{
             loader: 'coffee-loader',
             options: {
               literate: true
            }
         }]
       }]
     }
   }
   ```

   The code in the preceding example shows how the loader can be used and a new rule set up.

3. Should you need it, we will show you how to install `coffee-redux`. **Redux** is an open source library for managing JavaScript application states. It is frequently used with libraries such as **React** and **Angular**. To install it, type the following command:

   ```
   npm i -D coffee-redux-loader
   ```

The preceding example will not only help you understand the procedure for installing and using CoffeeScript in your bundles but also how the procedure works for the loaders that are not mentioned here, as they work in pretty much the same way.

You will see, however, that the shorthand installation and development mode for setting the command line has been used— `i` and `-D`, respectively. This works, in most cases, although you may find that there is now a response when there is a compatibility issue between your command-line utility and the Webpack version you are using.

Doing things this way can save you time, but when in doubt, use the verbose command-line convention demonstrated in this guide.

Now, let's move on to `coverjs`, which works a little differently.

coverjs

`coverjs` allows your code to be instrumented. This essentially means it allows your code to be measured or monitored in terms of performance.

The `coverjs` loader does not need to be combined with `mocha-loader`, as it is independent. The `reportHtml` function will append to the output of the body.

In the following example, `webpackOptions.js` is the subject of the code. In the first set of curly braces (`{`) are the options related to the module export procedures. In the double-set of braces (`[{`) is the code that binds the `coverjs` loader and the `test:""` statement (which indicates that every file will be tested):

```
webpack - dev - server "mocha!./cover-my-client-tests.js"--options
webpackOptions.js

// webpackOptions.js
module.exports = {
    output: "bundle.js",
    publicPrefix: "/",
    debug: true,
    includeFilenames: true,
    watch: true,
    postLoaders: [{
       test: "",
       exclude: [
         "node_modules.chai",
         "node_modules.coverjs-loader",
         "node_modules.webpack.buildin"
    ],
    loader: "coverjs-loader"
 }]
}

// cover-my-client-tests.js
require("./my-client-tests");
after(function() {
   require("cover-loader").reportHtml();
});
```

As you can see, this particular loader has its options set through a local file. Amendments to this will have the same effect as the configuration discussed in the previous chapter. That should be all you need to get `coverjs` up and running in your application. Next up is a more complicated topic, involving the use of international languages.

i18n-loader

`i18n-loader` deals with internationalization (`i18n`), which is the process of preparing an application so that it supports local languages and cultural settings. Let's set it up by going through the following steps:

1. Begin with the command-line installation:

   ```
   npm install i18n-loader
   ```

2. Now, let's start by using CSS. We will set up our style sheet for use with `i18n`. This is done with our usual project style sheet, `css/styles.css`. It's fine if you are importing another style sheet; just make the alterations there:

   ```
   . / colors.json {
           "red": "red",
           "green": "green",
           "blue": "blue"
       }
       . / de - de.colors.json {
           "red": "rot",
           "green": "green"
       }
   ```

   The loader can be called, assuming our locale is `de-de-berlin` (the German language and regional settings, for the sake of this example).

3. Next, we will localize the color scheme for our code in the `i18n` setting by using the `index.js` file:

   ```
   var locale = require("i18n!./colors.json");
   ```

   Now, wait for the status to be ready. This is only required once for all locales because all locales of the same language are merged into one module chunk:

   ```
   locale(function() {
    console.log(locale.red); // prints red
    console.log(locale.blue); // prints blue
   });
   ```

   This is usually all done in the same file as previously. The preceding code will append a child node to the `locale` variable in the `console.log` function, which will help with testing.

4. Now, configure the loader with `webpack.config.js` and make use of the related options available. As there are also options with this loader, you should tell the loader about all your locales, if you want to load them once and then want to use them synchronously:

   ```
   {
       "i18n": {
           "locales": [
               "de",
               "de-de",
               "fr"
           ],
           // "bundleTogether": false
       }
   }
   ```

   Note the `// "bundleTogether": false` statement in the preceding code—this can be uncommented and set to disable the bundling of locales.

   There are alternative calls. The following code chooses the correct file by locale:

   ```
   require("i18n/choose!./file.js");
   ```

However, this will not merge the objects. In the following code, the first line will concatenate all the locales that fit and the second line will merge the resulting objects:

```
require("i18n/concat!./file.js");
require("i18n/merge!./file.js");
```

As a result, `./file.js` in the preceding code block is executed while compiling:

```
require("i18n!./file.json") ==
   require("i18n/merge!json!./file.json")
```

The preceding code block simply reinforces the regular expression. It ensures that either of the `require` statements will load the same file.

Don't forget to polyfill `require` if you want to use it in Node.js. See the *Further reading* section at the end of this chapter for the relevant Webpack documentation.

The previous code blocks simply adjusted your project so that it is compatible with German-language territories with a German-speaking audience. Next up is `imports-loader`.

imports-loader

`imports-loader` allows you to use modules that depend on specific global variables. This is useful for third-party modules that may rely on global variables. `imports-loader` can add the necessary `require` calls, making them work with Webpack. Let's set it up by going through the following steps:

1. To install the loader in the command line, use the following statement:

   ```
   npm install imports-loader
   ```

   Assuming that you have the `example.js` file, this loader allows you to append an imported script to image tags using jQuery, as follows:

   ```
   $("img").doSomeAwesomeJqueryPluginStuff();
   ```

2. The `$` variable can then be injected into the module by configuring `imports-loader`, as follows:

   ```
   require("imports-loader?$=jquery!./example.js");
   ```

This will simply prepend `var $ = require("jquery");` to `example.js`. This can be useful, for instance, if you are optimizing your code to run libraries locally.

Using `polymer-loader` similarly optimizes your code or automates your processes to allow transitions. This is the next topic of our discussion.

polymer-loader

`polymer-loader` is used to convert HTML files into JavaScript files. To configure the loader, use the following code with `webpack.config.js`:

```
{
  test: /\.html$/,
  include: Condition(s) (optional),
  exclude: Condition(s) (optional),
  options: {
    ignoreLinks: Condition(s) (optional),
    ignorePathReWrite: Condition(s) (optional),
    processStyleLinks: Boolean (optional),
    htmlLoader: Object (optional)
  },
  loader: 'polymer-webpack-loader'
},
```

The `polymer-webpack-loader` phrase allows the developer to write HTML, CSS, and JavaScript code together as polymer elements in single documents—for instance, while still being able to use the full Webpack system, including module bundling and code splitting.

script-loader

`script-loader` essentially allows JavaScript to be loaded in a single instance. This applies globally throughout your project. Let's set this up by going through the following steps:

1. To install `script-loader`, type the following in the command line:

   ```
   npm install --save-dev script-loader
   ```

   Note that this doesn't work in Node.js.

2. Use `webpack.config.js` to configure Webpack, which will export `exec` from `'script.exec.js';`.:

```
module.exports = {
    module: {
        rules: [{
            test: /\.exec\.js$/,
            use: ['script-loader']
        }]
    }
}
```

There is also an inline way of doing this, as follows:

```
import exec from 'script-loader!./script.js';
```

That should be all you need to be able to use `script-loader` with your Webpack application. Next is `source-map-loader`.

source-map-loader

`source-map-loader` extracts existing source maps from all JavaScript entries in your project. This includes both inline source maps as well as those that are externally loaded. All source map data is processed as per a chosen source map style, which you specify with the `devtool` option in `webpack.config.js`. The following code shows that configuration:

```
module.exports = {
     module: {
     rules: [
     {
       test: /\.script\.js$/,
       use: [
   {
     loader: 'script-loader',
     options: {
           sourceMap: true,
              },
   },
        ]
      }
         ]
      }
  }
```

This loader can be extremely useful when using third-party libraries that have source maps. If not extracted and processed into the source map of the bundle, browsers may interpret source map data incorrectly. This loader allows the maintenance of source map data continuity across libraries and frameworks to ensure easy debugging.

The loader will extract from any JavaScript file, including those in the `node_modules` directory. Care should be taken when setting `include` and `exclude` rule conditions to optimize bundling performance.

less-loader

`less-loader` loads **LESS** (a type of **CSS**) scripts. You should install this in the usual way using the command line—for example, `npm i less-loader`. For the uninitiated, LESS is a more syntactically succinct form of CSS that is useful for backward compatibility.

You should chain `less-loader` with `css-loader` and `style-loader` to immediately apply all styles to the document. To configure this, use the following example code using the `webpack.config.js` file:

```
module.exports = {
    module: {
        rules: [{
            test: /\.less$/,
            use: [{
                    loader: 'style-loader', // creates style nodes
                                               from JS strings
                },
                {
                    loader: 'css-loader', // translates CSS
                                             into  CommonJS
                },
                {
                    loader: 'less-loader', // compiles Less to CSS
                },
            ],
        },
      ],
    },
  };
```

You can pass any LESS-specific options to the `less-loader` through the loader options. Since these options are passed to LESS as part of the program, they will need to be passed in `camelCase`; the following example in `webpack.config.js` shows how this is done:

```
module.exports = {
    module: {
        rules: [{
            test: /\.less$/,
            use: [{
                    loader: 'style-loader',
                },
                {
                    loader: 'css-loader',
                },
                {
                    loader: 'less-loader',
                    options: {
                        strictMath: true,
                        noIeCompat: true,
                    },
                },
            ],
          },
        ],
    },
  };
```

Note that LESS won't map all options to `camelCase` individually. It is recommended that you check the relevant executable and search for the `dash-case` option.

When in production mode, it is usually recommended that you extract the style sheets into a dedicated file using `MiniCssExtractPlugin`. This way, your styles are not dependent on JavaScript (more on this later).

In this section, we have discussed loaders in depth and also made a detailed examination of some of the more useful ones. Most loaders follow the same logic of installation and configuration and are built by the Webpack community, rather than by Webpack themselves. We'll discuss custom loaders in more detail in the final chapter of this book.

There are too many other loaders to mention here, but this gives you a very solid foundation to work with them in all sorts of imaginative ways. Something that is related to the non-native script that loaders tend to handle is the use of APIs with Webpack. We'll investigate this in the next section.

APIs

APIs are crucial to getting the most out of Webpack. In simple terms, APIs are required when communicating between applications and a website. When it comes to a JavaScript bundler such as Webpack, this includes the database and backend connectivity.

There is a huge number of APIs available when working with JavaScript but we cannot go through all of them here; however, there are some more useful common, but also complex, APIs that are used as tools quite often when programming with Webpack.

These tools are specific to Webpack and allow extensive or versatile functionality within Webpack, rather than just simple access to external code. The most notable of these tools are Babel and Node.js. So, let's use these tools as examples and learn about their usage in the following subsections, starting with Babel.

Babel and its loader builder

In case you are not aware, Babel is a toolchain that mainly converts ECMAScript 2015 and the preceding code into a regressive version of JavaScript that is compatible with current and older browsers or environments. The main things that Babel can do for you as a developer are as follows:

- Transform syntax
- Polyfill features that are missing in your target environment using `@babel/polyfill`
- Perform source code transformations, called code mods

Webpack uses an API for the Babel interface. If you attempt to install it with the command line, you should get the following message:

```
The Node.js API for babel has been moved to babel-core.
```

If you receive this message, it means that you have the `npm` package that Babel installed and it is using the short notation of the loader in the Webpack configuration file (which is not valid anymore, as of Webpack versions 2 and later).

In `webpack.config.js`, you may see the following script:

```
{
  test: /\.m?js$/,
  loader: 'babel',
}
```

As a result of installing Babel, Webpack tries to load the `babel` package instead of `babel-loader`.

To resolve this, the `npm` package Babel should be installed, as it is deprecated in Babel version 6. If you're using Babel version 6, then you should instead install `@babel/cli` or `@babel/core`. If one of your dependencies is installing Babel and you cannot uninstall it yourself, use the complete name of the loader in `webpack.config.js`:

```
{
  test: /\.m?js$/,
  loader: 'babel-loader',
}
```

The example you have followed so far should stand you in good stead for using Babel in general, but one key use of Babel is the customized loader. This is a topic covered more extensively in the final chapter of this book, but we will now go over how Babel works with customized loaders, especially since you may not use a loader that you customize yourself.

`babel-loader` exposes a `loader-builder` utility that allows users to add custom handling of Babel's configuration for each file that it processes.

The `.custom` phrase accepts a callback that will be called with the loader's instance of Babel. This is so that the tool can ensure that it uses the same `@babel/core` instance as the loader itself.

In cases where you want to customize but don't actually have a file to call `.custom`, you can also pass the `customize` option with a string pointing to a file that exports your custom callback function.

Probably the best way to learn how this works is through a practical example; let's go through one in the following exercise.

The goal of this example is to demonstrate how `babel-loader` can be used to build custom loaders.

This example first uses a custom filename. This can be whatever you want, but for the sake of this exercise, the name we have chosen is `./my-custom-loader.js`. You can export from this location or wherever you want:

1. Begin by creating a custom file by using the following code in `./my-custom-loader.js`:

```
module.exports = require("babel-loader").custom(babel => {
 function myPlugin() {
 return {
 visitor: {},
 };
 }
```

   In the preceding code block, we can see a `require` statement. This uses `babel-loader`, which we will need to create the custom loader.

2. Now, we need to configure our project to set up passing for the loader, as shown:

```
return {
    customOptions({
        opt1,
        opt2,
        ...loader
    }) {
        return {
            custom: {
                opt1,
                opt2
            },
            loader,
        };
    },
```

   Note that `custom:` refers to pulling out any custom options that the loader might have. Also, note the `loader` reference after the two options. This passes the options back with the two `custom` options removed.

3. Then, we pass Babel's `PartialConfig` object, setting the normal configuration as `return cfg.options`:

```
config(cfg) {
    if (cfg.hasFilesystemConfig()) {
        return cfg.options;
    }
```

```
            return {
                ...cfg.options,
                plugins: [
                    ...(cfg.options.plugins || []),
                    testPlugin,
                ],
            };
        },
```

In the preceding code, where the `testPlugin` statement is made, we can see the inclusion of a custom plugin, which will then be available as an option.

4. Now, let's make the placeholder text to test the custom loader on. The preceding code should generate something like the following:

```
result(result) {
return {
    ...result,
    code: result.code + "\n// Generated by this custom loader",
    };
    },
    };
});
```

This code block shows that the custom loader is generating code.

5. Ensure that your configuration is made correctly. Note that you should always replace `__dirname` and `custom-loader` with a name of your choice. In the Webpack configuration module, type the following:

```
.exports = {
    module: {
        rules: [{
            loader: path.join(__dirname, 'custom-loader.js'),
        }]
    }
};
customOptions(options: Object): {
    custom: Object,
    loader: Object
}
```

The preceding block shows you how to set up and configure `customOptions`.

6. Given the loader's options, split the custom options out of the options for `babel-loader`:

   ```
   config(cfg: PartialConfig): Object
   ```

7. Given Babel's `PartialConfig` object, return the `options` object that should be passed to `babel.transform`:

   ```
   result(result: Result): Result
   ```

Both of the preceding code blocks refer to the contents of the custom file we have built, which, in this example, is `./my-custom-loader.js`.

Note that Babel's `result` object will allow loaders to make additional tweaks to it.

That should be all you need to get the custom loader working with Babel. Read the last chapter of this book on authoring and custom loaders for further information. Another key API often used in Webpack projects is the Node.js API.

The Node.js API

The Node.js API is useful when using a custom development process. This is because all reporting and error handling is done manually. In this situation, Webpack simply handles the compilation processes. Note that the `stats` configuration options will not have any effect when using the Node.js API.

This API will be installed whenever you install Webpack 5; you can refer to the first chapter if you are reading this section out of sequence. Let's set this API up by taking the following steps:

1. Begin by including the `webpack` module to your Node.js script. This is done using the `webpack.config.js` file:

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

   webpack({
     // Configuration Object
   }, (some, stats) => { // Stats Object
     if (some || stats.hasErrors()) {
       // Handle errors here
     }
     // Done processing
   });
   ```

In the previous example, a callback function was provided—`webpack()`—which runs the compiler. The code presents a few conditions. This is only an example and should be substituted with your code, of course. Also, the `some` term should similarly be replaced with the correct object name associated with your project.

Note that the `some` object won't include compilation errors, but only issues related to Webpack specifically, such as misconfiguration. Those errors are instead handled using the `stats.hasErrors()` function.

2. Next, ensure that the passing of the compiler instance is done correctly.

If the Webpack function is not provided with a callback, it will return a `compiler` instance. The `compiler` instance can manually trigger the `webpack()` function or ensure that it watches for changes (using `.run(callback)` or `.watch(watchOptions, handler)`) during a build, or even runs the build itself without the need for the CLI.

The `compiler` instance permits the use of child compilers and will delegate all the bundling, writing, and loading work to registered plugins.

There is something called a `hook` property, which is part of the `compiler` instance. Its purpose is to register any plugin to any hook event during a compiler's life cycle. You can configure this compiler with the `WebpackOptionsDefaulter` and `WebpackOptions Apply` utilities.

Upon completion of a build run, the previously mentioned callback function will be executed. The final logging of any errors or statistics are done with this function.

The Node.js API will only support a single compilation once. Concurrent watches or builds can corrupt the output bundle.

Using the API to call a run is similar to using the `compiler` instance.

3. We should now run a compilation using `webpack.config.js`:

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

const compiler = webpack({
 // Configuration Object
});
```

```
compiler.run((some, stats) => { // Stats Object
});
```

4. From here, we can also trigger a `watch` session. When the `webpack()` function detects a change, it will run again and return an instance of `watching`:

```
watch(watchOptions, callback);
const webpack = require('webpack');

const compiler = webpack({
 // Configuration Object
});

const watching = compiler.watch({
 // Example watchOptions
 aggregateTimeout: 300,
 poll: undefined
}, (some, stats) => { // Stats Object
 // Print watch/build result here...
 console.log(stats);
});
```

As filesystem inaccuracies can trigger multiple builds if a change is detected, the `console.log` statement in the previous code block may cause this trigger multiple times for any single modification. Checking `stats.hash` can help you see whether the file has changed.

5. Using the `watch` method this way will return a `watching` instance and a `.close(callback)` method. The calling of this method will end the `watching` session:

```
watching.close(() => {
 console.log('Watching Ended.');
});
```

Note that using the `invalidate` watching function will allow the manual invalidation of the current compilation without stopping the `watch` process. This is useful as only one run is permitted at once:

```
watching.invalidate();
```

As multiple, simultaneous compilations are restricted, Webpack offers something called `MultiCompiler` to expedite your project development. It is a module that allows Webpack to run multiple configurations in separate compilers. If your `options` parameter is an array, Webpack will apply separate compilers and execute any callbacks after all the compilers have completed their process:

```
var webpack = require('webpack');
webpack([
 { entry: './index1.js', output: { filename: 'bundle1.js' } },
 { entry: './index2.js', output: { filename: 'bundle2.js' } }
], (some, stats) => { // Stats Object
 process.stdout.write(stats.toString() + '\n');
})
```

The preceding code block shows you how to configure your project in `webpack.config.js` to allow this procedure.

As explained, attempting to run these compilations in parallel will produce incorrect outputs. If this is done accidentally, error management comes to the fore.

Generally, error handling encompasses three types of errors—fatal Webpack errors (such as misconfiguration), compilation errors (such as missing assets), and compilation warnings.

The following code block shows you how you can configure your project—in this example, using `webpack.config.js`—to handle those errors:

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

webpack({
 // Configuration Object
}, (some, stats) => {
 if (some) {
   console.error(some.stack || some);
 if (some.details) {
   console.error(some.details);
 }
 return;
 }
const info = stats.toJson();
if (stats.hasErrors()) {
  console.error(info.errors);
 }
if (stats.hasWarnings()) {
  console.warn(info.warnings);
```

```
    }
  // Log results...
  });
```

In the preceding block, the `some` element indicates these errors as a variable. We can see the various conditions that will register those errors in the console log.

We have given you an intensive crash course on how APIs work with Webpack, so if you survived, you are now a hardcore expert in programming skills. Well done!

Now that we have explored a variety of loaders and APIs (including Babel and Node.js), it is time to look at the final feature covered in this chapter—plugins.

Plugins

Plugins serve the purpose of doing everything that a loader cannot do. Loaders often help run code that is not native to Webpack and plugins do the same; however, loaders are often built by the community whereas plugins are built by Webpack's in-house developers.

Plugins are said to be the backbone of Webpack. The software is built on the same plugin system that you use in your Webpack configuration.

Webpack has a rich plugin interface and most of the features within Webpack itself use it. The following is a bulleted list of the available plugins that this section will cover in detail. Next to each name is a brief description of the plugin:

- `BabelMinifyWebpackPlugin`: Minificates with `babel-minify`
- `CommonsChunkPlugin`: Extracts common modules shared between chunks
- `ContextReplacementPlugin`: Overrides the inferred context of a require expression
- `HotModuleReplacementPlugin`: Enables **Hot Module Replacement** (**HMR**) (more on this later)
- `HtmlWebpackPlugin`: Easily creates HTML files to serve your bundles
- `LimitChunkCountPlugin`: Sets minimum/maximum limits for chunking to better control the process
- `ProgressPlugin`: Reports compilation progress
- `ProvidePlugin`: Uses modules without having to use `import/require`
- `TerserPlugin`: Enables control of the version of Terser in your project

There are many more plugins available on the Webpack community pages; however, the preceding list illustrates the more salient and useful ones. Next, we will go over each one with a detailed explanation.

We won't go over the final three plugins as they only need to be installed in the usual fashion.

The installation of each plugin follows the same process as the one used for loaders:

1. First, install the plugin using the command line and then alter the configuration file to require the plugin, similar to the following example:

```
npm install full-name-of-plugin-goes-here-and-should-be-hyphenated-
and-not-camelcase --save-dev
```

   Remember, this is a generic example; you will have to add your plugin name. The same can be said of the following configuration, which follows the same procedure as most Webpack projects and the same configuration file that we previously used, `webpack.config.js`.

2. We should prepare our configuration file now:

```
const MinifyPlugin = require("full-name-of-plugin-goes-here-and-
should-be-hyphenated-not-camelcase");
module.exports = {
 entry: //...,
 output: //...,
 plugins: [
 new MinifyPlugin(minifyOpts, pluginOpts)
 ]
}
```

That concludes your general introduction to plugins. We took a general approach to prevent over-complication. Now, we can move on to some of the intriguing aspects of the various plugins available on Webpack. In the following subsection, we will discuss the following:

- `BabelMinifyWebpackPlugin`
- `CommonsChunkPlugin`
- `ContextReplacementPlugin`
- `HtmlWebpackPlugin`
- `LimitChunkCountPlugin`

Most plugins are developed by Webpack inhouse and fill the development gaps that loaders are not yet able to. These are some of the more interesting ones. Let's start with `BabelMinifyWebpackPlugin`.

BabelMinifyWebpackPlugin

In this subsection, we are going to install `BabelMinifyWebpackPlugin`. This plugin is designed to minify Babel script. Minifying, as previously discussed, refers to the removal of erroneous or surplus code to compress the application size. To use `babel-loader` and include `minify` as a preset, use `babel-minify`. The use of `babel-loader` with this plugin will be much faster and will operate on smaller file sizes.

Loaders in Webpack operate on single files and the preset of `minify` will execute each file directly in the browser's global scope; this is done by default. Some things in the top-level scope will not be optimized. Optimization of the `topLevel` scope of the file can be done using the following code and `minifyOptions`:

```
mangle: {
topLevel: true
}
```

When `node_modles` is excluded from the run through `babel-loader`, then the minify optimizations are not applied to the excluded files since nothing is passed through `minifer`.

When using `babel-loader`, the generated code does not go through the loader and is not optimized.

A plugin can operate on the entire chunk or bundle output and can optimize the whole bundle, and some differences can be seen in the minified output; however, in that case, the file size would be huge.

The Babel plugin is very useful when working with Webpack and it works across multiple platforms. The next plugin we will discuss is `CommonsChunkPlugin`, which is designed to work with multiple module chunks—again, something that is very native to Webpack.

CommonsChunkPlugin

`CommonsChunkPlugin` is an optional feature that creates a separate feature known as a chunk. Chunks consist of common modules shared between multiple entry points.

This plugin has been removed in Webpack 4 (Legato).

Looking into `SplitChunkPlugin` will show you more on how chunks are treated in Legato.

The resulting chunked file can be initially loaded once by separating the common modules that form the bundle. This is stored in the cache for later use.

Using the cache in this way will allow the browser to load pages quicker, rather than forcing it to load larger bundles.

Use the following declaration in your configuration to use this plugin:

```
new webpack.optimize.CommonsChunkPlugin(options);
```

A short and simple installation, but worth the introduction. Next, let's work with `ContextReplacementPlugin`.

ContextReplacementPlugin

`ContextReplacementPlugin` refers to a `require` statement with an extension, such as the following:

```
require('./locale/' + name + '.json')
```

When you encounter an expression such as this, the plugin will infer the directory of `./local/` and a regular expression. If a name is not included at the time of compilation, then every file will be included as a module in the bundle. This plugin will allow the inferred information to be overridden.

The next plugin to discuss is `HtmlWebpackPlugin`.

HtmlWebpackPlugin

`HtmlWebpackPlugin` simplifies the creation of HTML files to serve the bundle. This is especially useful for bundles that include a hash in the filename, which changes every compilation.

When using this method, we have three choices—use a template with `lodash` templates, use your loader, or let the plugin generate an HTML file. The template is simply an HTML template that you can automatically load using `lodash` for speed. The loader or the plugin can generate its HTML file. This all speeds up any automated processes.

When using multiple entry points, they will be included with the `<script>` tags in the generated HTML.

If you have any CSS assets in Webpack's output, then these will be included with the `<link>` tags in the `<head>` element of the generated HTML. An example would be if CSS is extracted with `MiniCssExtractPlugin`.

Back to working with chunks. The next plugin to look at is a plugin that deals with limiting the chunk count.

LimitChunkCountPlugin

`LimitChunkCountPlugin` is used when loading on-demand because many code splits can be used. When compiling, you might notice that some chunks are very small, which creates a larger HTTP overhead. `LimitChunkCountPlugin` can postprocess chunks by merging them.

Limiting the maximum number of chunks can be done by using a value greater than or equal to `1`. Using `1` prevents additional chunks from being added as the main or entry point chunk:

```
[/plugins/min-chunk-size-plugin]
```

Keeping the chunk size above the specified limit is not a feature you will find in Webpack 5; `MinChuckSizePlugin` should be used, instead, in this situation.

That concludes our introduction to plugins and this chapter in general. Plugins allow Webpack to work in a variety of ways and Webpack enables developers to build loaders and plug gaps in functionality issues. They are indispensable when working on a larger scale project or complex projects that require automation.

Our section on APIs showed you how, sometimes, we don't always want to use local code and provided you with an excellent transition to libraries, which we will discuss in the next chapter.

Summary

This chapter has given you an in-depth view of loaders and how they are used in Webpack. Loaders are fundamental to Webpack, but plugins are the core and backbone. This chapter took you through the major features of these topics, demonstrating the best uses for each and when it is a good time to switch between them.

We then explored plugins and how they are used, including Babel, custom plugins, and loaders. We have also looked at APIs and their usage, specifically APIs that facilitate a wider range of functionality within Webpack, such as the Babel and Node APIs.

In the next chapter, we will discuss libraries and frameworks. Our examination of plugins, APIs, and loaders has revealed that sometimes, we don't want to use remote code such as libraries, but other times we do. Webpack generally deals with locally hosted code, but there are times when we might want to work with libraries. This provides a good transition to that topic.

Questions

1. What is an `i18n` loader?
2. What is the toolchain that Webpack typically uses to convert ECMA scripts?
3. What is Babel mainly used for?
4. Which loader allows users to add custom handling of Babel's configuration?
5. What does `polymer-webpack-loader` do?
6. What does `polymer-webpack-loader` offer a developer?
7. When using the Node.js API, what will the callback function that is provided run?

5
Libraries and Frameworks

This chapter looks at how applications work with libraries and frameworks. Numerous frameworks and libraries work with Webpack. Typically, these are JavaScript frameworks. They are becoming an increasingly central part of programming and knowing how to integrate them into your application bundle will likely be an increasing need.

Working with libraries and frameworks is a little bit different from working with other Webpack elements. By following typical examples and use cases, this book will explore Angular and how to structure the Angular framework to facilitate package bundling. This includes what to expect from the Webpack bundling, the desired outcomes, the advantages, and the limitations.

Once you have completed this chapter, you should be confident in how to use these major frameworks and libraries with Webpack. You will also know how to integrate and install them, as well as how to use best practices in your integrations.

In this chapter, we will cover the following topics:

- Best practices
- Working with JSON
- Working with YAML
- Working with Angular
- Working with Vue.js

Best practices

So far, we've only covered building Vanilla JavaScript, which should not be confused with the Vanilla framework. Even though it is the best way to learn, it is more likely that you will work with some kind of framework. Webpack will work with any JavaScript or TypeScript framework, including Ionic and jQuery; however, more tricky frameworks include Angular, Vue, and YAML.

Now, we will get to grips with YAML, but before delving into this, you may be wondering whether backend frameworks can be integrated. The simple answer is that they can, but they are not bundled. The only level of integration, however, is through linking source code, as we do with most projects or APIs, such as REST APIs.

As we have already discussed, Webpack has a production and development mode. Production mode bundles your project into its finished state, ready for web delivery or publication, and provides little room for tweaking. Development mode gives the developer the freedom to modify database connections; this is how the backend integrates. Your project's backend will likely be **ASP.NET** or **PHP**, but some backends are more complex and utilize `OpenAuth`. As a developer, you will need an overview of all of them. This guide, however, only deals with Webpack.

Rest assured that all of these frameworks will integrate, which is done via a REST API, which returns data in **JavaScript Object Notation** (**JSON**) format. It is also possible to do this using **AJAX**. In any case, ensure that secure best practices are followed, as JSON calls to databases are not as secure as using a server-side script.

If your project uses Ionic, then you should follow the instructions for Angular as the Ionic framework is based on this.

That should give you a solid overview of the best practices for working with backends and libraries. We will now discuss each of the most common libraries that you will encounter in a Webpack. Let's start with JSON, as it is the easiest to understand and is the most important way that external or backend code and databases interact with your Webpack application.

Working with JSON

Most of the time, when you work with frameworks, you will need to communicate across languages and between applications. This is done with JSON. JSON works similarly to YAML in this way, but it is much easier to understand how Webpack works with JSON, first.

JSON files are understood by Webpack's compiler without the need for a dedicated loader and, therefore, can be considered a native script to Webpack's bundler, just as JavaScript is.

As alluded to in this guide so far, JSON files are instrumentally used for package composition. How Webpack records and tracks the use of loaders and dependencies is through a schema in the form of a JSON file. This is typically the `package.json` file, or sometimes the `package.lock.json` file, which records the exact version of each package installed so that it can be reinstalled. In this context, "package" refers to loaders and dependencies, collectively.

Each JSON file must be programmed correctly or it won't be readable by Webpack. Unlike JavaScript, comments are not permitted in code to guide users, so you may want to use a `README` file to explain its content and purpose to the user.

The structure of a JSON file is a little different from JavaScipt and contains a different array of elements, such as keys. Take the following code block as an example:

```
module: {
    rules: [{
    test: /\.yaml$/,
    use: 'js-yaml-loader',
    }]
}
```

This is an extract from the `package.json` file that we'll use a little later on in this chapter. The content of this block essentially declares the parameters of a module. The named module is used as a key with a colon after it. These keys are sometimes referred to as names and they are where options are placed. This code sets a series of rules, and in this rule is the instruction to make Webpack use `js-yaml-loader` whenever it transpiles the content modules.

You must ensure that braces and brackets are used in the right sequence or Webpack won't be able to read the code.

As a JavaScript developer, you may be very familiar with JSON and how it is used. However, it's worth spelling out in case there are any blind spots. YAML is a more complex framework but it is something you will often encounter, so it's only progressively more complex than JSON. Let's get to grips with it now.

Working with YAML

YAML is a common framework that is used in a similar way to JSON. The difference is that YAML is more commonly used with configuration files, which is why you may encounter it more frequently, or for the first time, when using Webpack.

To use YAML with Webpack, we must install it. Let's get started:

1. You can install YAML using `npm` in your command-line utility. Use the following command:

   ```
   yarn add js-yaml-loader
   ```

   Note the `yarn` statement. This refers to an open source package manager for JavaScript files that comes preinstalled with `Node.js`. It works similarly to `npm` and should be preinstalled. If you don't get a response from the code used here, then double-check that it is preinstalled.

2. To inspect your YAML files from the command-line interface, you should install them globally:

   ```
   npm install -g js-yaml
   ```

3. Next, open up the configuration file, `webpack.config.js`, and make the following amendments:

   ```
   import doc from 'js-yaml-loader!./file.yml';
   ```

   The preceding line will return a JavaScript object. This method is safe for data that is not trusted. See the *Further reading* section for a GitHub YAML example.

4. After that, use `webpack.config.js` to configure the file to allow the use of the YAML loader:

   ```
   module: {
       rules: [{
       test: /\.yaml$/,
       use: 'js-yaml-loader',
       }]
   }
   ```

   You may also want to use the YAML front-matter loader for Webpack. This is an MIT-licensed piece of software that will convert YAML files into JSON, which will be particularly useful if you are more used to using the latter. If you are a JavaScript developer, it is quite likely that you are used to using JSON as it tends to be more commonly used with JavaScripters than YAML.

   This module requires a minimum of Node v6.9.0 and Webpack v4.0.0 installed on your device. Webpack 5 is the subject of this guide, so there should be no trouble there. However, note that this feature is only available with Webpack 4 and 5.

The following steps are separated from the previous steps as they deal with the installation of `yaml-loader` and not `yaml-frontmatter`, which is used to convert YAML into JSON files (which is more typical of Webpack package structures):

1. To begin, you'll need to install `yaml-frontmatter-loader` using your command-line utility:

   ```
   npm install yaml-frontmatter-loader --save-dev
   ```

   This particular command line may be syntactically different from the kind that this guide has shown in the past, but regardless of the format, this command should work.

2. Then, add the loader to your configuration, as follows:

   ```
   const json = require('yaml-frontmatter-loader!./file.md');
   ```

   This code will return the `file.md` file as a JavaScript object.

3. Next, open `webpack.config.js` once more and make the following changes to the `rules` key, ensuring you reference the YAML loader:

   ```
   module.exports = {
     module: {
       rules: [
         {
            test: /\.md$/,
            use: [ 'json-loader', 'yaml-frontmatter-loader' ]
         }
       ]
     }
   }
   ```

4. Next, run Webpack 5 via your preferred method and see the results!

If you got through that in one piece, you may be feeling brave enough to tackle Angular. That is a more difficult framework to work with, so let's get started.

Working with Angular

Angular is a library and framework, and as with all frameworks, it aims to make building applications easier. Angular utilizes dependency injection, integrated best practices, and end-to-end tooling, all of which can help resolve development challenges.

Angular projects often use Webpack. At the time of writing, the latest version of Angular in use is **Angular 9**. An updated version is brought out every year.

Now, let's take a look at how Webpack works when bundling Angular projects or even bolting Angular on to existing projects.

Angular looks for `window.jQuery` to determine whether jQuery is present. See the following code block for the source code:

```
new webpack.ProvidePlugin({
  'window.jQuery': 'jquery'
});
```

To add a `lodash` map, append the existing code with the following:

```
new webpack.ProvidePlugin({
  _map: ['lodash', 'map']
});
```

Webpack and Angular work by supplying Webpack with entry files and letting it incorporate the dependencies that lead from those entry points. The entry point file in the following example is the root file of the application—`src/main.ts`. Let's take a look:

1. We need to use the `webpack.config.js` file here. Note that this is a single entry-point process:

   ```
   entry: {
    'app': './src/main.ts'
   },
   ```

   Webpack will now parse the file, inspect it, and traverse its imported dependencies recursively.

2. Make the following changes in `src/main.ts`:

   ```
   import { Component } from '@angular/core';
   @Component({
    selector: 'sample-app',
    templateUrl: './app.component.html',
    styleUrls: ['./app.component.css']
   })
   export class AppComponent { }
   ```

Webpack will recognize that the `@angular/core` file is being imported, so this will be added to the dependency list for bundle inclusion. Webpack will open the `@angualar/core` file and trace its series of `import` statements until a dependency graph is built from it. This will be built from `main.ts` (a TypeScript file).

3. These files will then be provided as output to the `app.js` bundle file that is identified in the configuration:

```
output: {
 filename: 'app.js'
}
```

The JavaScript file that contains the source code and dependencies is a single file and the output bundle is the `app.js` file. This will be loaded later with a JavaScript tag (`<script>`) in the `index.html` file.

It is advised that you don't have one giant bundle for everything, for obvious reasons. Therefore, it is recommended that the volatile application code is separated from the more stable vendor code modules.

4. This separation of the application and vendor code is done by changing the configuration so that two entry points are now used—`main.ts` and `vendor.ts`—as shown:

```
entry: {
 app: 'src/app.ts',
 vendor: 'src/vendor.ts'
},
output: {
 filename: '[name].js'
}
```

Two bundle files are emitted from Webpack by constructing two separate dependency graphs. The first is called `app.js`, while the second is called `vendor.js`. Each contains the application code and vendor dependencies, respectively.

In the preceding example, `file name: [name]` indicates a placeholder that is replaced with entry names by the Webpack plugin, app, and vendor. Plugins are covered in more detail in the next chapter, so if you're stuck, maybe mark this page and come back to it.

5. Now, instruct Webpack of which parts belong to the vendor bundle by adding a `vendor.ts` file that only imports third-party modules, as in the following code, which shows an example of the contents of `vendor.ts`:

```
import '@angular/platform-browser';
import '@angular/platform-browser-dynamic';
import '@angular/core';
import '@angular/common';
import '@angular/http';
import '@angular/router';
// RxJS
import 'rxjs';
```

   Note the mention of `rxjs`. This is a library for reactive programming that aims to make it easier for developers to compose asynchronous code or callbacks.

Other vendors can be imported this way, such as jQuery, Lodash, and Bootstrap. File extensions that can also be imported include **JavaScript** files (`.js`), **TypeScript** files (`.ts`), **Cascading Style Sheets** files (`.css`), and **Syntactically Awesome Style Sheets** files (`.sass`).

Angular can be a very complicated framework and is very relevant to web-based applications. However, your particular need may suit single-page applications better, in which case Vue.js would be the preferred option to use for most. Let's take a look at it now.

Working with Vue.js

Vue.js is another framework but is an open source. The salience of its use, or area of distinct purpose, lies within **single-page applications** (**SPAs**). This is because the framework focuses on delivering a seamless experience, but with fewer features than Angular, which can work alongside many other languages and still operate very quickly. Building applications that are quite large with Vue.js will result in very slow loading during use and even slower compilation.

Perhaps the best way to understand this is to consider jQuery and how it uses inline statements to call in a script to pages, whereas Angular uses core modules, each designed with a specific purpose. Vue.js lies somewhere in between the pure and simple jQuery for SPAs and Angular.

Using Webpack with the Vue.js project is done with the use of a dedicated loader.

The installation of `vue-loader` and `vue-template-compiler` is advised, unless you're an advanced user of Vue.js's template compiler. Follow these steps:

1. To follow this example, begin by installing `vue-loader` and `vue-template-compiler` with the following code:

   ```
   npm install -D vue-loader vue-template-compiler
   ```

   The template compiler has to be installed separately so that the version can be specified.

   Vue.js has released a corresponding version of its template compiler with each new release. The two versions must be in sync so that the loader produces code that is runtime compatible. So, every time you upgrade one, you should upgrade the other.

   The loader associated with Vue.js has a slightly different configuration from most loaders. Make sure you add the Vue.js loader's plugin to your Webpack configuration when handling files with extensions of `.vue`.

2. This is done by altering the configuration of Webpack, shown in the following example using `webpack.config.js`:

   ```
   const VueLoaderPlugin = require('vue-loader/lib/plugin')
   module.exports = {
    module: {
    rules: [
    // other rules
    {
      test: /\.vue$/,
      loader: 'vue-loader'
     }
    ]
   },
    plugins: [
    // make sure to include the plugin.
    new VueLoaderPlugin()
    ]
   }
   ```

   This plugin is required as it is responsible for cloning riles that are defined and applying them to language blocks that correspond to `.vue` files.

3. Using Vue.js, we add the following content:

   ```
   new webpack.ProvidePlugin({ Vue: ['vue/dist/vue.esm.js', 'default']
   });
   ```

The preceding code must be added as it contains the full installation of the ECMAScript module for Vue.js when used with bundlers. This should be present in the `/dist` folder of your project's `npm` package. Note that `.ems.` signifies the ECMAScript module. There are runtime-only and production-specific installation methods shown on the Vue.js installation page, which is available in the *Further reading* section of this chapter. **UMD** and **CommonJS** installations are similar and use the `vue.js` and `vue.common.js` files, respectively.

As our project will use the `.esm` format, it may be useful to know more about it. It has been designed to be analyzed statically, which allows bundlers to perform **tree-shaking**, which is the elimination of unused code. Note that the default file for bundlers is `pkg.module`, which is responsible for runtime-only ECMAScript module compilation. For more information, see the Vue.js installation page—the URL is available in the *Further reading* section of this chapter.

This concludes the content of this chapter regarding frameworks and libraries. You should now be in a strong position to work with complex projects that may even utilize more than one framework.

Summary

This chapter covered typical frameworks and how to get started with using them. This included the installation process that should be followed and what criteria and peripherals are needed. This guide has paid attention to best practices and security. When you begin your project, you should follow these examples ahead of time, paying close attention to procedures, warnings, and requirements.

This guide has given you an overview of other frameworks, such as RxJS for callbacks and jQuery, as well as pointed you in the right direction when it comes to using unusual file extensions. We have also explored the usage and installation procedures for Angular's core functionality and Vue.js when working with Webpack 5 and how Vue.js is better suited to single-page applications and how Angular works better on larger projects.

Having covered most of the core topics, in the next chapter we will delve into deployment and installation. This will be even more important when working with databases and ensuring security requirements are kept. The next chapter will provide an in-depth account of this subject and will hopefully address any concerns you might have as a developer.

Further reading

- GitHub's YAML example: `https://github.com/nodeca/js-yaml`
- The window.jQuery source code: `https://github.com/angular/angular.js/blob/v1.5.9/src/Angular.js#L1821-L1823`
- The Vue.js installation guide: `https://vuejs.org/v2/guide/installation.html`

Questions

1. What compiler needs to correspond to the version of `Vue.js` being used?
2. When using Angular, this guide advises the separation of volatile code and stable vendor code. This is done using two entry points. What are they?
3. What are the minimal installation requirements when using YAML with Webpack?
4. Why should you install your YAML files globally?
5. What is an SPA?
6. Where should you add Vue.js's loader when handling `.vue` files?
7. What is missing from the following line of code:
   `import 'angular/http';`
8. When using Angular, how is `app.js` loaded?
9. What is YARN?
10. What is the default `pkg.module` file used for?

6
Production, Integration, and Federated Modules

In this chapter, we will cover production, integration, and federated modules. We will provide an overview of the correct deployment procedures, shortcuts, and alternatives. Even though some of this chapter will discuss subjects that have already been covered in more detail, it's good to go over them again so that you have a clearer understanding of what you've learned so far.

So far, we have discussed and carried out development builds and alluded to going into production, but the procedure for appropriate publication-level production is a little different and involves cross-checking and following best practices.

This section of the book will explore the various options we can use to deploy Webpack 5 with various web utilities. This will provide you with an overview of such web utilities and explain which are more appropriate for specific situations and platforms, including deploying with Babel.

All of these subjects are relevant to our opening section on production bundling, which also covers the topic of deployment for production and publication purposes.

In this chapter, we will cover the following topics:

- Production setup
- Shimming
- Progressive web applications
- Integrating task runners
- GitHub
- Extracting boilerplate
- Module Federation

Production setup

Now that we understand the basics, we can move on and learn about how to practically deploy production bundles. The objectives of building a project in development mode and production mode differ greatly. In production mode, goals shift to minifying builds using lightweight source mapping and optimizing assets to improve load time. In development mode, strong source mapping is crucial, as well as having a **localhost** server with live reloading or hot module replacement. Due to their different purposes, it is typically recommended to write separate Webpack configurations for each mode.

A common configuration should be maintained between modes, despite their differences. To merge these configurations, a utility called `webpack-merge` can be used. This common configuration process means code does not need to be duplicated with each mode. Let's get started:

1. Begin by opening your command-line utility, installing `webpack-merge`, and saving it in development mode, as follows:

   ```
   npm install --save-dev webpack-merge
   ```

2. Now, let's examine the **project directory**. Its contents should be structured similar to the following:

   ```
   webpack5-demo
   |- package.json
   |- webpack.config.js
   |- webpack.common.js
   |- webpack.dev.js
   |- webpack.prod.js
   |- /dist
   |- /src
     |- index.js
     |- math.js
   |- /node_modules
   ```

   Note that the preceding output shows that there are extra files present in this particular example. For instance, we are including the `webpack.common.js` file here.

3. Let's take a closer look at the `webpack.common.js` file:

```
const path = require('path');
const { CleanWebpackPlugin } = require('clean-webpack-plugin');
const HtmlWebpackPlugin = require('html-webpack-plugin');

module.exports = {
  entry: {
    app: './src/index.js'
  },
  plugins: [
    // new CleanWebpackPlugin(['dist/*']) for < v2 versions of
    // CleanWebpackPlugin
    new CleanWebpackPlugin(),
    new HtmlWebpackPlugin({
      title: 'Production'
    })
  ],
  output: {
    filename: '[name].bundle.js',
    path: path.resolve(__dirname, 'dist')
  }
};
```

The `webpack.common.js` file deals with **CommonJS** requests. It does things similarly to ECMAScript but is formatted differently. Let's ensure that the `webpack.config.js` file, which works in the **ECMA** environment, does the same thing as the **CommonJS** configuration file. Take note of the entry points and bundle name, as well as the `title` option. This latter option is the mode's counterpart, so you must ensure there is parity between both files in your project.

4. Here, we're looking inside the `webpack.dev.js` file:

```
const merge = require('webpack-merge');
const common = require('./webpack.common.js');

module.exports = merge(common, {
  mode: 'development',
  devtool: 'inline-source-map',
  devServer: {
    contentBase: './dist'
  }
});
```

As you can see, the preceding code provides instructions on how `webpack.common.js` should be used in development mode. This is simply a case of cross-checking for final production to ensure the content of your work is formatted correctly and will map with an error during compilation.

5. If you're working in production mode, the following file, `webpack.prod.js`, will be called into action:

```
const merge = require('webpack-merge');
const common = require('./webpack.common.js');

module.exports = merge(common, {
  mode: 'production',
});
```

Using `webpack.common.js`, set up entry and output configurations and include any plugins that are required for both environment modes. When using `webpack.dev.js`, the mode should be set to development mode. Also, add the recommended **devtool** to that environment, as well as the simple `devServer` configuration. In `webpack.prod.js`, the mode is, of course, set to production, which loads `TerserPlugin`.

Note that `merge()` can be used in an environment-specific configuration so that you can easily include a common configuration in development and production modes. It is also worth noting that there are a variety of advanced features available when using the **`webpack-merge`** tool.

6. Let's make those development configurations inside of `webpack.common.js`:

```
{
 "name": "development",
 "version": "1.0.0",
 "description": "",
 "main": "src/index.js",
 "scripts": {
 "start": "webpack-dev-server --open --config webpack.dev.js",
 "build": "webpack --config webpack.prod.js"
},
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "clean-webpack-plugin": "^0.1.17",
    "css-loader": "^0.28.4",
    "csv-loader": "^2.1.1",
            "express": "^4.15.3",
            "file-loader": "^0.11.2",
            "html-webpack-plugin": "^2.29.0",
            "style-loader": "^0.18.2",
            "webpack": "^5.0.0",
            "webpack-dev-middleware": "^1.12.0",
            "webpack-dev-server": "^2.9.1",
            "webpack-merge": "^4.1.0",
            "xml-loader": "^1.2.1"
        }
    }
```

The preceding example simply shows the completed configuration for CommonJS. Note the list of plugin dependencies and their versions, which are loaded through the `devDependancies` option.

7. Now, run those scripts and see how the output changes.
8. The following code shows how you can continue adding to the production configuration:

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

Note that Webpack 5 will minify the project's code by default when in production mode.

`TerserPlugin` is a good place to start minification and should be used as the default option. There are, however, a couple of choices, such as `BabelMinifyPlugin` and `closureWebpackPlugin`.

When trying a different minification plugin, ensure that the choice will also drop any dead code, similar to tree shaking, which we described earlier in this book. Something related to tree-shaking is shimming, which we'll discuss next.

Shimming

Shimming, or more specifically, `shim-loaders`. Now is a good time to explore this concept in detail since you'll need a firm grasp of it before you can move on.

The compiler that Webpack uses can understand modules written in **ECMAScript 2015**, **CommonJS**, or **AMD**. It should be noted that some third-party libraries may need global dependencies, for example, when using jQuery. These libraries may, in this case, need globals to be exported. This almost broken nature of the module is where shimming comes into effect.

Shimming can allow us to convert one language specification into another. In Webpack, this is usually done through dedicated loaders specific to your environment.

The principal concept behind Webpack is the use of modular development – isolated modules that are securely contained that don't rely on hidden or global dependencies – so it is important to note that the use of such dependencies should be sparse.

Shimming can be useful when you wish to polyfill browsers so that you can support multiple users. In this case, the polyfill only needs to be patched where needed and loaded on demand.

Shimming is related to patching but tends to take place in the browser, which makes it highly relevant to progressive web applications.

In the next section, we will look at progressive web applications in more detail – they are key to Webpack 5.

Progressive web applications

Sometimes called PWAs, progressive web applications deliver a native application experience online. They have many contributing factors, the most notable of which is the ability for the application to function when offline, as well as online. To do this, service workers are utilized.

Allowing your web app to work offline will mean you can provide functionality such as push notifications. These rich experiences will also be available to the web-based application through devices such as the service worker. This script will work in the background of a browser, regardless of whether the user is on the right page or not, and permit these same notifications or even background synchronization.

PWAs offer the reach of the web but are fast and reliable, similar to desktop applications. They can also feel engaging, like mobile apps, and can offer the same immersive experience. This demonstrates a new level of quality for applications. They can also play a part in cross-platform compatibility.

Responsive design was the web's first push in this direction, but the push to make the internet more universal has led us to PWA. To leverage your application's potential, you should use a progressive approach. For more information, see this Google resource on the subject: `https://developers.google.com/web/progressive-web-apps/`.

When using Webpack, the service workers need to be registered so that you can start integrating desktop functionality into your web-based PWA. PWAs are not designed to be installed locally by the user; they work natively through web browsers.

Service workers can be registered by adding the following to your code – in this example, this is an `index.js` file:

```
if ('serviceWorker' in navigator) {
window.addEventListener('load', () => {
  navigator.serviceWorker.register('/service-
    worker.js').then(registration => {
     console.log('SW registered: ', registration);
  }).catch(registrationError => {
   console.log('SW registration failed: ', registrationError);
  });
});
}
```

With this complete, run `npm build` – you should see the following output in the command-line window:

```
SW registered
```

Now, the application can be served with `npm start` in the command-line interface.

A PWA should have a manifest, a service worker, and possibly a workbox to wrap and protect the service worker. For more information on the manifest, see `Chapter 3`, *Using Configurations and Options*. Workbox is a plugin that can be installed in the command line using the following command:

```
npm install workbox-webpack-plugin --save-dev
```

An example configuration of the **Workbox** plugin can be seen here in a hypothetical `webpack.config.js` file:

```
const WorkboxPlugin = require('workbox-webpack-plugin');
new WorkboxPlugin.GenerateSW({
 clientsClaim: true,
 skipWaiting: true,
 }),
```

The options inside the `{` braces will encourage the service workers to get to that point quickly and will not allow them to strangle previous service workers.

As your project becomes more complicated, you may consider using related task runners. Let's take a look at this in more detail.

Integrating task runners

Task runners handle automation tasks such as code linting. Webpack is not designed for this purpose – no bundler is. However, using Webpack, we can benefit from the high level of focus that task runners offer while still having high-performance bundling.

While there is some overlap between task runners and bundlers, they can integrate well if approached correctly. In this section, we will explore integration techniques we can use for some of the most popular task runners out there.

Bundlers work by preparing JavaScript and other frontend code for deployment, transforming their format so that they're suitable for browser delivery. For example, it allows us to split JavaScript into chunks for lazy loading, which improves performance.

Bundling may be challenging, but its results will remove a lot of the painstaking work of the process.

In this section, we will show you how to integrate the following:

- Gulp
- Mocha
- Karma

Gulp is probably the most well-known task runner, so let's start with that first. It is utilized through the use of a dedicated file, much like the other two task runners. Here, the `gulpfile.js` file will handle how Webpack interplays with Gulp. Let's take a look at how to integrate all of these task runners:

1. First, let's take a look at the `gulpfile.js` file:

```
const gulp = require('gulp');
const webpack = require('webpack-stream');
gulp.task('default', function() {
   return gulp.src('src/entry.js')
    .pipe(webpack({
      // Any configuration options...
    }))
    .pipe(gulp.dest('dist/'));
});
```

   This is all we have to do to use Gulp. Note the use of the `gulp.task` function, the `return` entry point, and the `.pipe(gulp.dest('dist/'));` output location.

2. The following is the command-line code you can use to install Mocha:

   ```
   npm install --save-dev webpack mocha mocha-webpack
   mocha-webpack 'test/**/*.js'
   ```

   For more information, please visit the Webpack community repository.

3. The following are the configuration file amendments you'll need to make to the `karma.config.js` file to use Karma with Webpack:

   ```
   module.exports = function(config) {
     config.set({
       files: [
         { pattern: 'test/*_test.js', watched: false },
         { pattern: 'test/**/*_test.js', watched: false }
       ],
       preprocessors: {
         'test/*_test.js': [ 'webpack' ],
         'test/**/*_test.js': [ 'webpack' ]
       },
       webpack: {
         // Any custom webpack configuration...
       },
       webpackMiddleware: {
         // Any custom webpack-dev-middleware configuration...
       }
     });
   };
   ```

The `webpack` and `webpackMiddleware` options have been left blank so that you can fill this content with your project's specific configuration. These options can be completely removed if you are not using them. For the sake of this example, we won't be.

These installation procedures will be some use to you if you wish to use them in your development environment, but GitHub is an increasingly more essential tool. We'll take a look at how it can play a key role in terms of development in the next section.

GitHub

GitHub, as you may already know, is a command-line code hosting platform that works well with Webpack. Much of the code and projects you will work with when using Webpack will be hosted through GitHub.

GitHub is based on the Git version control system. The use of GitHub with Webpack 5 allows some command-line operations to be used online.

A Git command-line instruction is typically made with the use of the `git` command before each new entry and before each command. Much of Webpack's content files are available through GitHub, and the GitHub Webpack page can be found here: `https://github.com/webpack/webpack`. The development of Webpack 5 can be viewed here in terms of stages of progress, which can be interesting and allows you to anticipate its arrival better, should you need to upgrade your projects. The URL for this is `https://github.com/webpack/webpack/projects/5.`

As a developer, you may have used GitHub often, but if you're a dedicated JavaScript developer, you may have limited experience. When working on Webpack projects, the GitHub platform offers a great deal of live and collaborative opportunities. Since version control and command-line functionality is provided, there is less of a need to perform software development locally. This is the primary reason why GitHub is so popular among the developer community and why it is becoming so fundamental as proof of a developer's work.

GitHub allows developers to work together on projects. When working with bundled projects, this is even more useful as some command-line operations can be run online. GitHub also allows Agile workflows or project management interfaces. Agile methodologies allow teams to collaborate while individuals self-organize through a dedicated digital platform.

When using GitHub, you may be working with other people's code. This may include frameworks of code that have been developed by teams. This can become very difficult, even for the most seasoned developers, if they're unfamiliar with the logic being used. This brings us to the subject of boilerplate, which is usually standard or well-documented code, but nonetheless, you may want to extract this from the sections of a project you wish to utilize. This is where this extraction process starts to become very useful.

Extracting boilerplate

Boilerplate code is sections of code that need to be included in various places, but with little or no alterations made to them. When using languages that are considered verbose, it is often necessary to write exhaustive amounts of code. This large section of code is called boilerplate. It has essentially the same purpose as a framework or library, and the terms are often conflated or mutually acceptable.

The Webpack community offers boilerplate installations, such as the combined installation of multiple common dependencies such as prerequisites and loaders. There are multiple versions of these boilerplates, and the use of these can expedite a build. Please search the Webpack Community pages (`https://webpack.js.org/contribute/`) or Webpack's GitHub page (`https://github.com/webpack-contrib`) for examples.

This being said, there will be times when only part of the boilerplate is required. For this, extracting the boilerplate functionality of Webpack may be required.

Webpack, while using its minification method, allows boilerplate to be extracted; that is, only the elements of the boilerplate you need are included in the bundle. This is an automated process that's done during the compilation process.

Minification is the key way in which Webpack 5 offers this extraction process and it's one of the more salient ways in which a bundler of this type can be used. There is another key process that is highly useful and native to Webpack 5. It takes us in a different direction regarding how bundles are built but is something that will no doubt follow on from a complex or custom build, such as a project that began by boilerplate being extracted. This process is known as Module Federation.

Module Federation

Module Federation has been described as a game-changer in the JavaScript architecture. It essentially allows applications to run code from remotely stored modules between servers while part of a bundled application.

Some developers may be aware of a technology called **GraphQL**. It is essentially a solution for sharing code between applications, developed by a company called Apollo. Federated Modules are a feature of Webpack 5 that allows this to happen between bundled applications.

For a long time, the best compromise was the use of externals of the `DllPlugin`, which relied on a centralized external dependency file, However, this isn't great for organic development, convenience, or large-scale projects.

With Module Federation, JavaScript applications can dynamically load code between applications and share dependencies. If an application is using a federated module as part of its build but requires a dependency to serve the federated code, Webpack can also download that dependency from the origin of the federated build. So, the federation will effectively provide a map of where Webpack 5 can find the required dependency code.

There is some terminology to consider when it comes to federation: remote and host. The term remote refers to the application or modules that are loaded into the user's application, while the host refers to the application that the user is visiting through their browser at runtime.

The federation method is designed for standalone builds and can be deployed independently or in your own repository. In this sense, they can be hosted bi-directionally, effectively serving as the host of remote content. This means that a single project could potentially switch between hosting orientations throughout the user's journey.

Building our first federated application

Let's start by looking at three standalone applications, identified as the first, second, and third applications, respectively.

The first application in our system

Let's begin by configuring the first application:

1. We are going to be using the `<app>` container in the HTML. This first app is a remote application in the federation system and is, therefore, to be consumed by other applications.

2. To expose the application, we will use the `AppContainer` method:

```
const HtmlWebpackPlugin = require("html-webpack-plugin");
const ModuleFederationPlugin =
   require("webpack/lib/container/ModuleFederationPlugin");

module.exports = {
  plugins: [
   new ModuleFederationPlugin({
    name: "app_one_remote",
    remotes: {
      app_two: "app_two_remote",
      app_three: "app_three_remote"
 },
 exposes: {
   'AppContainer':'./src/App'
 },
 shared: ["react", "react-dom","react-router-dom"]
 }),
 new HtmlWebpackPlugin({
   template: "./public/index.html",
   chunks: ["main"]
  })
 ]
}
```

This first application will also consume a component from two other federated applications in the system.

3. To allow this application to consume components, the remote scope will need to be specified.
4. All these steps should be followed as specified in the preceding code block.
5. Now, let's take a look at the HTML segment of the build:

```
<head>
  <script src="http://localhost:3002/app_one_remote.js"></script>
  <script src="http://localhost:3003/app_two_remote.js"></script>
</head>
<body>
  <div id="root"></div>
</body>
```

The preceding code shows the `<head>` element of the HTML. `app_one_remote.js` connects the runtime and provisional orchestration layer at runtime. It is specifically designed for entry points. These are example URLs and you can use your own locations. It's important to note that this example is a very low-memory example and that your build may be much larger, but it is good enough to understand this principle.

6. To consume code from a remote application, the first application has a web page that consumes a dialog component from the second application, as follows:

```
const Dialog = React.lazy(() => import("app_two_remote/Dialogue"));
const Page1 = () => {
  return (
    <div>
      <h1>Page 1</h1>
        <React.Suspense fallback="Loading User Dialogue...">
          <Dialog />
        </React.Suspense>
    </div>
  );
}
```

7. Let's start by exporting the default HTML page we are using and set up the router, which is done as follows:

```
import { Route, Switch } from "react-router-dom";
import Page1 from "./pages/page1";
import Page2 from "./pages/page2";
import React from "react";
   const Routes = () => (
     <Switch>
       <Route path="/page1">
        <Page1 />
       </Route>
       <Route path="/page2">
        <Page2 />
       </Route>
     </Switch>
  );
```

The preceding code block shows how the code works; it will export the default routes from each page in the system we are developing.

This system is part of three applications, the second of which we will look at now.

The second application

The system that we're building is comprised of three applications. This application will expose the dialogue that enables the first application in this sequence to consume it. The second application, however, will consume the first application's `<app>` element identifier. Let's take a look:

1. We will start by configuring the second application. This means we need to specify `app-one` as a remote application and simultaneously demonstrate bi-directional hosting:

```
const HtmlWebpackPlugin = require("html-webpack-plugin");
const ModuleFederationPlugin =
  require("webpack/lib/container/ModuleFederationPlugin");
module.exports = {
  plugins: [
    new ModuleFederationPlugin({
    name: "app_two_remote",
    library: { type: "var", name: "app_two_remote" },
    filename: "remoteEntry.js",
    exposes: {
      Dialog: "./src/Dialogue"
```

```
     },
       remotes: {
         app_one: "app_one_remote",
     },
       shared: ["react", "react-dom","react-router-dom"]
     }),
     new HtmlWebpackPlugin({
       template: "./public/index.html",
       chunks: ["main"]
      })
     ]
    };
```

2. For the sake of consumption, the following is what the root application looks like:

```
import React from "react";
import Routes from './Routes'
const AppContainer = React.lazy(() =>
  import("app_one_remote/AppContainer"));

const App = () => {
  return (
   <div>
     <React.Suspense fallback="Loading App Container from Host">
       <AppContainer routes={Routes}/>
     </React.Suspense>
   </div>
  );
}
```

3. Next, we need to set up the code so that we can export the default application. The following is an example of what the default page should look like, when using the dialogue:

```
import React from 'react'
import {ThemeProvider} from "@material-ui/core";
import {theme} from "./theme";
import Dialog from "./Dialogue";

function MainPage() {
   return (
     <ThemeProvider theme={theme}>
       <div>
         <h1>Material User Iinterface App</h1>
         <Dialog />
```

```
          </div>
        </ThemeProvider>
    );
  }
```

4. Now, we need to export the default `MainPage`. This is done using the third application in our system.

The third application

Let's take a look at our third and final application:

1. The third application in our system will be exporting a default `MainPage`. This is done through the following script:

```
new ModuleFederationPlugin({
   name: "app_three_remote",
   library: { type: "var", name: "app_three_remote" },
   filename: "remoteEntry.js",
   exposes: {
     Button: "./src/Button"
 },
 shared: ["react", "react-dom"]
}),
```

As expected, the third application looks similar to the previous ones, except it doesn't consume `<app>` from the first application. This application is standalone and has no navigation and as such, does not specify any remote federated components.

When viewing the system in a browser, you should pay close attention to the network tab. The code can be federated across three different servers (potentially) and three different bundles (naturally).

This component permits a great deal of dynamism in your builds, but you may want to avoid federating the whole application container unless you hope to utilize **server-side rendering (SSR)** or progressive loading as, otherwise, loading times could be severely harmed.

Loading issues are a natural concern, but one issue that would ordinarily cause larger project sizes is the potential duplication of repetitious code, which is the result of using multiple parallel bundles. Let's take a look at how Webpack 5 handles that issue.

Duplication issues

One key feature of Webpack is removing duplicated code. In the federation environment, host applications serve the remote application with dependencies. In the absence of a sharable dependency, the remote application can download its own automatically. This is a kind of built-in redundancy fail-safe mechanism.

Manually adding vendor code can be tedious at scale but the federation feature allows us to create automation scripts. This is the developer's choice, but it may be an excellent opportunity for you to be able to test your knowledge.

We have already alluded to SSR. You should be aware that server builds require a `commonjs` library target so that they can be used with Webpack federation. This can be done with S3 streaming, ESI, or by automating npm publish so that you can consume server variants. The following code shows an example of including `commonjs`:

```
module.exports = {
 plugins: [
  new ModuleFederationPlugin({
    name: "container",
    library: { type: "commonjs-module" },
    filename: "container.js",
    remotes: {
      containerB: "../1-container-full/container.js"
 },
   shared: ["react"]
  })
 ]
};
```

You may wish to use the `target: "node"` method to avoid URLs in favor of file paths. This will allow SSR with the same code base but different configurations when building for Node.js. This will also mean that separate builds will become separate deployments.

Webpack, as a company, is open to demonstrating SSR examples that you, as part of a community of developers, may have made. They will gladly accept pull requests through their GitHub pages since they have the bandwidth to and benefit from the exposure while the process is so new.

Summary

In this chapter, you looked at the process you might use to deploy a project online. We went over the installation and setup process, as well as tree shaking.

First, we looked at the production and development modes, the nature of each environment, and how they utilize Webpack. Then, we looked at shimming, the best way to use it, how it works so that we can patch code, its relationship with task runners, and their integration with bundlers such as Webpack.

Now, you should be able to extract boilerplate, integrate various task runners, and know how to use GitHub.

In the next chapter, we will discuss hot module replacement, and live coding, and get to grips with some serious tutorials.

Questions

1. What is meant by boilerplate?
2. What does tree shaking do?
3. What is meant by the term shimming?
4. What is the purpose of progressive web applications?
5. What do task runners do?
6. What three task runners were mentioned in this chapter?
7. Webpack's compiler can understand modules written in which three specifications?

Further reading

- Webpack content files and the GitHub Webpack page can be found here: `https://github.com/webpack/webpack`.
- Webpack 5 can be viewed here in terms of its stages of progress: `https://github.com/webpack/webpack/projects/5`.
- Webpack community pages: `https://webpack.js.org/contribute/`
- Webpack's GitHub pages: `https://github.com/webpack-contrib`

7
Debugging and Migration

This chapter will look further into migration and debugging, providing an extensive overview and detailed inspection of these topics.

Migration refers to the process of moving content and projects from earlier versions of Webpack to newer ones. We will specifically look at the move from Webpack version 3 to version 4 and from version 4 to version 5. We will also cover how to deal with deprecated plugins and how to remove or update them. This will include a look at migration when using Node.js v4 and the **command-line interface** (**CLI**).

This chapter will discuss the `resolve` method and how `module.loaders` has now been replaced by the `module.rules` method. It will also cover the chaining of loaders, including chaining loaders that are no longer needed or have been removed.

Then, this chapter will move on to exploring debugging. Debugging involves the process of removing common faults and errors that present themselves in complex software systems. This chapter will explain common problems and their solutions, troubleshooting, the best practices to follow to avoid these problems, and how to find faults.

Topics covered in this chapter are as follows:

- Debugging
- Hot module replacement
- Adding a utility
- Migration

Debugging

Debugging tools are central to a workflow, particularly when contributing to core reproduction, writing a loader, or any other complex form of coding. This guide will take you through the utilities that are of most use when figuring out matters such as slow performance or an unforgiving traceback. These principle utilities are as follows:

- The `stats` data made available through Node.js and the CLI
- Chrome DevTools via `node-nightly` and the latest Node.js versions

In Webpack 5, as of the time of writing, there are some known problems; for example, DevTools doesn't support persistent caching and persistent cache files that include absolute paths are not yet portable.

The `stats` data can be very useful when debugging build issues, sifting through data manually, or when using a tool. It can be used to find the following:

- Build errors and warnings
- The contents of every module
- Module compilation and resolving stats
- The interrelationships between modules
- The modules contained within any given chunk

Also, the official Webpack **analyze** tool will accept this data and visualize it for you.

Sometimes, a more robust solution is needed when console statements simply won't do the job.

As is commonly asserted among the community of frontend developers, Chrome DevTools is indispensable when debugging applications—but it doesn't stop there. As of Node.js v6.3.0+, the built-in **inspection flag** can be used by developers to debug a Node.js program in DevTools.

This short demonstration will utilize the `node-nightly` package, which provides access to the latest inspection capabilities. This offers the ability to create breakpoints, debug memory usage issues, expose objects in the console, and more:

1. Begin by installing the `node-nightly` package globally:

```
npm install --global node-nightly
```

2. This package must now be run using the command line to finish the installation:

```
node-nightly
```

3. Now, using the `inspect` flag feature of `node-nightly`, we can start to debug any Webpack project. It should be noted that `npm` scripts cannot be run now; instead, the full `node_module` path will need to be expressed:

```
node-nightly --inspect ./node_modules/webpack/bin/webpack.js
```

4. The output should reveal something like the following in the command-line utility window:

```
Debugger listening on ws://127.0.0.1:9229/c624201a-250f-416e-
a018-300bbec7be2c For help see https://nodejs.org/en/docs/inspector
```

5. Now, moving to Chrome's inspection feature (`chrome://inspect`), any active scripts should now be viewable under the `Remote Target` header.

   Clicking the `inspect` link under each script will open a dedicated debugger or DevTools link for the node in a session, which will connect automatically. Note that **NiM** is a handy extension for Chrome that will automatically open DevTools in a new tab every time you make an inspection. This is extremely useful for longer projects.

   It may also be useful to use the `inspect-brk` flag, which causes a break on the first statement of any script, allowing the source code to be perused, the breakpoints to be set, and the process to be stopped and started ad hoc. This also allows the programmer to continue to pass arguments to the script in question; this may be useful for making parallel configuration alterations.

One key feature that this all relates to—and something that has been alluded to previously in this guide—is the exciting subject of **hot module replacement** (HMR). What it is and how to use it will be covered in the following section, along with a tutorial.

Hot module replacement

HMR is possibly the most useful element of Webpack. It allows runtime updates of modules that need a total refresh. This section will explore the implementation of HMR, as well as detailing how it works and why it is so useful.

It is very important to note that HMR is not intended for and should never be used in production mode; it should only be used in development mode.

It's worth noting that, according to the developers, the internal HMR API for plugins will probably change in future updates of Webpack 5.

To enable HMR, what we first need to do is update our `webpack-dev-server` configuration and use Webpack's built-in HMR plugin. This feature is great for productivity.

It is also a good idea to remove the entry point for `print.js` as it will now be consumed by the `index.js` module.

Anyone who used `webpack-dev-middleware` instead of `webpack-dev-server` should now use the `webpack-hot-middleware` package to enable HMR:

1. To start using HMR, we need to return to the configuration file, `webpack.config.js`. Follow the amendment here:

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

module.exports = {
  entry: {
     app: './src/index.js',
     print: './src/print.js'
     app: './src/index.js'
  },
  devtool: 'inline-source-map',
  devServer: {
    contentBase: './dist',
    hot: true
  },
  plugins: [
    // new CleanWebpackPlugin(['dist/*']) for < v2 versions of
    // CleanWebpackPlugin
    new CleanWebpackPlugin(),
    new HtmlWebpackPlugin({
      title: 'Hot Module Replacement'
    }),
    new webpack.HotModuleReplacementPlugin()
  ],
  output: {
    filename: '[name].bundle.js',
    path: path.resolve(__dirname, 'dist')
  }
};
```

Note the additions in the preceding code—the `hot:` option is set to `true` and the `'Hot Module Replacement'` plugin has been added—as well as the creation of the new Webpack plugin in the configuration for HMR. All of this should be done to make use of the plugin and HMR.

2. The command line can be used to modify the `webpack-dev-server` configuration with the following command:

```
webpack-dev-server --hot
```

This will allow ad hoc changes to be made to bundled applications.

3. `index.js` should now be updated so that when a change in `print.js` is detected, Webpack can accept the updated module. The changes are illustrated in bold in the following example; we are simply exposing the `print.js` file with an `import` expression and function:

```
import _ from 'lodash';
import printMe from './print.js';

function component() {
  const element = document.createElement('div');
  const btn = document.createElement('button');

  element.innerHTML = _.join(['Hello', 'Webpack'], ' ');

  btn.innerHTML = 'Click me and check the console!';
  btn.onclick = printMe;

  element.appendChild(btn);

  return element;
}

document.body.appendChild(component());

if (module.hot) {
  module.hot.accept('./print.js', function() {
    console.log('Accepting the updated printMe module');
    printMe();
    })
  }
```

If you make changes to the console log in `print.js`, the following output will be seen in the browser console. The obligatory `printMe()` button is missing for now, but that can be updated later:

```
export default function printMe() {
  console.log('This got called from print.js!');
  console.log('Updating print.js...')
}
```

A look at the console window should reveal the following printout:

```
 [HMR] Waiting for update signal from WDS...
main.js:4395 [WDS] Hot Module Replacement enabled.
  2main.js:4395 [WDS] App updated. Recompiling...
  main.js:4395 [WDS] App hot update...
  main.js:4330 [HMR] Checking for updates on the server...
  main.js:10024 Accepting the updated printMe module!
  0.4b8ee77….hot-update.js:10 Updating print.js...
  main.js:4330 [HMR] Updated modules:
  main.js:4330 [HMR]  - 20
```

The preceding block reveals that HMR is awaiting a signal from Webpack and should HMR take place, the command-line utility can perform the automatic bundle amendment. The command-line window will also show this when left open. Node.js has an API that can be used in a similar way.

Using DevServer with the Node.js API

When using **DevServer** and the Node.js API, you should not put the `dev server` option on the Webpack configuration object; instead, it should always be passed as a secondary parameter.

Here, DevServer simply refers to the use of Webpack in development mode as opposed to the `watching` or `production` modes. To use DevServer with the Node.js API, follow these steps:

1. The function is placed in the `webpack.config.js` file, as follows:

```
new WebpackDevServer(compiler, options)
```

To enable HMR, the configuration object must first be modified to include the HMR entry points. The `webpack-dev-server` package includes a method called `addDevServerEntryPoints` that can be used to do this.

2. What follows is a short example of what it might look like using `dev-server.js`:

```
const webpackDevServer = require('webpack-dev-server');
const webpack = require('webpack');

  const config = require('./webpack.config.js');
  const options = {
    contentBase: './dist',
    hot: true,
    host: 'localhost'
};

webpackDevServer.addDevServerEntrypoints(config, options);
const compiler = webpack(config);
const server = new webpackDevServer(compiler, options);

server.listen(5000, 'localhost', () => {
  console.log('dev server listening on port 5000');
});
```

HMR can be difficult. To demonstrate this, in our example, click the button that has been created in the example web page. It is evident that the console is printing the old function. This is because the event handler is bound to the original function.

3. To resolve this for use with HMR, the binding must be updated to the newer function using `module.hot.accept`. See the following example using `index.js`:

```
import _ from 'lodash';
import printMe from './print.js';

function component() {
  const element = document.createElement('div');
  const btn = document.createElement('button');

  element.innerHTML = _.join(['Hello', 'Webpack'], ' ');

  btn.innerHTML = 'Click me and view the console!';
  btn.onclick = printMe;  // onclick event is bind to the
                          // original printMe function
  element.appendChild(btn);

  return element;
}

document.body.appendChild(component());
let element = component(); // Store the element to re-render on
                           // print.js changes
document.body.appendChild(element);

if (module.hot) {
  module.hot.accept('./print.js', function() {
    console.log('Accepting the updated printMe module!');
    printMe();
    document.body.removeChild(element);
    element = component();
    document.body.appendChild(element);
  })
}
```

By way of explanation, `btn.onclick = printMe;` is an `onclick` event that is bound to the original `printMe` function. `let element = component();` will store the element to re-render on any changes to `print.js`. Also, note the `element - component();` statement, which will re-render the component and update the click handler.

This is just one example of the kind of pitfalls that you may encounter. Luckily, Webpack offers a lot of loaders, some of which are discussed later, that make HMR much less problematic. Let's now look at HMR and style sheets.

HMR and style sheets

Using HMR with CSS is a little more straightforward with the help of `style-loader`. This loader uses `module.hot.accept` to patch style tags when CSS dependencies are updated.

In the next stage of our practical example, we will be taking the following steps:

1. Start by installing both loaders with the following command:

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

2. Next, update the configuration file, `webpack.config.js`, to make use of the loaders:

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

module.exports = {
  entry: {
    app: './src/index.js'
  },
  devtool: 'inline-source-map',
  devServer: {
    contentBase: './dist',
    hot: true
  },
  module: {
    rules: [
      {
        test: /\.css$/,
        use: ['style-loader', 'css-loader']
      }
    ]
  },
  plugins: [
    // new CleanWebpackPlugin(['dist/*']) for < v2 versions of
      // CleanWebpackPlugin
    new CleanWebpackPlugin(),
    new HtmlWebpackPlugin({
      title: 'Hot Module Replacement'
    }),
    new webpack.HotModuleReplacementPlugin()
  ],
  output: {
    filename: '[name].bundle.js',
    path: path.resolve(__dirname, 'dist')
  }
};
```

Hot loading style sheets is as easy as importing them into the module, as you can see from the text in bold in the previous configuration example and the directory structure example to follow.

3. Ensure that you organize the project files and directories, as shown, in the following structure:

```
webpack5-demo
| - package.json
| - webpack.config.js
| - /dist
  | - bundle.js
| - /src
  | - index.js
  | - print.js
  | - styles.css
```

4. Append the style sheet by adding a `body` style to style the background of the document body associated with it blue. Do this using the `styles.css` file:

```
body {
  background: blue;
}
```

5. After that, we need to ensure the content is loaded to the `index.js` file correctly, as follows:

```
import _ from 'lodash';
import printMe from './print.js';
import './styles.css';

function component() {
  const element = document.createElement('div');
  const btn = document.createElement('button');

  element.innerHTML = _.join(['Hello', 'Webpack'], ' ');

  btn.innerHTML = 'Click me and check the console!';
  btn.onclick = printMe;  // onclick event is bind to the
                          // original printMe function

  element.appendChild(btn);

  return element;
}

let element = component();
document.body.appendChild(element);

if (module.hot) {
  module.hot.accept('./print.js', function() {
    console.log('Accepting the updated printMe module!');
```

```
        document.body.removeChild(element);
        element = component(); // Re-render the "component" to update
                               // the click handler
        document.body.appendChild(element);
      })
    }
```

Now, when the style of the `body` tag background class is changed to red, the color change should be immediately noted without a page refresh, indicating the live nature of hot coding.

6. You should now make these changes to the background using `styles.css`:

```
body {
  background: blue;
  background: red;
}
```

This demonstrates, in a very simple way, how live code edits can be made. This was only a simple example, but it serves as a good introduction. Now, let's progress to something trickier—loaders and frameworks.

Other loaders and frameworks

The numerous loaders that are available and that we alluded to earlier make HMR interact more smoothly with a variety of frameworks and libraries. Some of the more useful ones are described here:

- **Angular HMR**: A simple change to your main `NgModule` file is all that's required to have full control over the HMR APIs (does not require a loader).
- **React Hot Loader**: This loader tweaks React components in real time.
- **Elm Hot Webpack Loader**: This loader supports HMR for the Elm programming language.
- **Vue Loader**: This loader supports HMR for Vue components out of the box.

We have gone through HMR and related loaders and frameworks but something we have yet to discuss—but relates to the content we have covered so far—is adding a utility. We will get to grips with that in the following section.

Adding a utility

In this context, a utility means a file or module that is responsible for a related set of functions, designed to optimize, analyze, configure, or maintain. This is in contrast to an application, which tends to perform a task or set of tasks aimed directly at users. Therefore, you may think of a utility, in this context, as something that is part of the frontend but is hidden away in the background for background tasks.

To begin, add a utility file to the example project. Do this in `src/math.js` so that it exports two functions:

1. The first step will be to organize the project directory:

   ```
   webpack5-demo
   |- package.json
   |- webpack.config.js
   |- /dist
     |- bundle.js
     |- index.html
   |- /src
     |- index.js
     |- math.js
   |- /node_modules
   ```

   The **project tree** shows how your files and folder should look and you will note some new additions in there, such as `math.js`.

2. Let's now take a closer look at how `math.js` is coded:

   ```
   export function square(x) {
     return x * x;
   }

   export function cube(x) {
     return x * x * x;
   }
   ```

   You'll see that they are simple-to-export functions; they will come to the fore later.

3. Also, make sure you set the Webpack mode to `development` in the configuration file, `webpack.config.js`, which ensures the bundle is not minified:

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

   module.exports = {
     entry: './src/index.js',
   ```

```
    output: {
      filename: 'bundle.js',
      path: path.resolve(__dirname, 'dist')
    }
    },
    mode: 'development',
    optimization: {
      usedExports: true
    }
  };
```

4. With this in place, we next update the entry script to utilize one of these new methods and remove `lodash` for simplicity. This is done using the `src/index.js` file:

```
    import _ from 'lodash';
    import { cube } from './math.js';

    function component() {
      const element = document.createElement('div');
      const element = document.createElement('pre');

      // Lodash, now imported by this script
      element.innerHTML = _.join(['Hello', 'webpack'], ' ');
      element.innerHTML = [
        'Hello Webpack!',
        '5 cubed is equal to ' + cube(5)
      ].join('\n\n');

      return element;
    }

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

From the proceeding example, we can see that the `square` method was not imported from the `src/math.js` module. This function can be considered dead code—essentially, an unused export that can be dropped.

5. Now, run an `npm` build again to inspect the results:

```
npm run build
```

6. Once that is done, locate the `dist/bundle.js` file—it should be somewhere on lines 90–100. Search the file for code similar to the following example to follow this process:

```
/* 1 */
/***/ (function(module, __webpack_exports__, __webpack_require__) {
  'use strict';
  /* unused harmony export square */
  /* harmony export (immutable) */ __webpack_exports__['a'] = cube;
  function square(x) {
    return x * x;
  }

  function cube(x) {
    return x * x * x;
  }
});
```

   In this example, you will now see an `unused harmony export square` comment. Note that it is not being imported. It is, however, still included in the bundle for the time being.

7. ECMA scripting is not perfect, so it is important to provide hints to Webpack's compiler about the purity of the code. The `packages.json` property will help with these side effects:

```
{
  "name": "your-project",
  "sideEffects": false
}
```

   The preceding code doesn't contain side effects; therefore, the property should be marked as `false` to instruct Webpack to remove the unused exports.

   In this context, a side effect is defined as a script that performs a special behavior when imported, as opposed to exposing more than one export, and so on. An example would be **polyfills**, which affect the global project and usually do not provide an export.

8. In the event of the code having a side effect, an array can be provided as a remedy, such as in the following example:

```
{
  "name": "your-project",
  "sideEffects": [
    "./src/some-side-effectful-file.js"
  ]
}
```

The array in this example accepts relative and absolute patterns.

9. Note that any imported file is subject to tree shaking. For example, if `CSS-loader` is used to import a CSS file, it must be added to the side effects list to prevent it from being unintentionally dropped in production mode:

```
{
  "name": "your-project",
  "sideEffects": [
    "./src/some-side-effectful-file.js",
    "*.css"
  ]
}
```

10. Finally, `sideEffects` can also be set from the `module.rules` configuration option. So, we've queued up our dead code to be dropped by using the `import` and `export` syntax, but we still need to drop it from the bundle. To do that, set the `mode` configuration option to `production`. This is done by appending the configuration file, `webpack.config.js`, as follows:

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

module.exports = {
  entry: './src/index.js',
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist')
  },
  mode: 'development',
  optimization: {
    usedExports: true
  }
  mode: 'production'
};
```

The `--optimize-minimize` flag can also be used to enable `TerserPlugin`. Now that we have understood that, we can run another `npm` build.

It will now be clear that the whole bundle is minified and mangled. A closer look reveals that the `square` function is missing; instead, you have a mangled cube function:

```
function r(e){return e*e*e}n.a=r
```

With minification and tree shaking, our bundle is now a few bytes smaller! While that may not seem like much in this contrived example, tree shaking can yield a significant decrease in bundle size when working on larger applications with complex dependency trees.

`ModuleConcatenationPlugin` is needed for the tree shaking to work. It is added by using `mode: "production"`. If you are not using it, remember to add `ModuleConcatenationPlugin` manually.

The following tasks must be done to take full advantage of tree shaking:

- Use the ES2015 module syntax (that is, `import` and `export`).
- Ensure no compilers transform your ECMAScript syntax into CommonJS modules.
- Add a `sideEffects` property to your `package.json` file.
- Use the `production` configuration option to enable various optimizations, including tree shaking and minification.

When it comes to tree shaking, it often helps to think of your application as a tree. In this analogy, the source code and libraries would be the green leaves and the living part of the tree, respectively. Dead code, however, would represent the dead leaves. Shaking that tree will remove the now defunct code.

This is especially relevant and worth considering when migrating. Given the changes in code deprecation between versions of Webpack, it is important to get your software working at its best before attempting anything like this. This will prevent very difficult bugs from developing.

Migration

Migration relates to moving from one version of Webpack to another. This usually involves upgrading to the latest version. It's something that, as a web developer, you may already know is tricky when dealing with other software in general, so this section will be an important one that perhaps you can refer back to when going through future development.

For the sake of providing a more detailed guide, a migration strategy for moving from Webpack 3.0 to Webpack 4.0 is included, so let's go through this now, before moving on to version 5.

Prerequisites when migrating to version 4 from version 3

There are several prerequisites to go over before we begin migrating our project from Webpack version 3 to 4. These involve the following:

- Node.js
- The CLI
- Plugins
- Modes

For developers using Node.js version 4 or lower, upgrading to Node.js version 6 or higher is necessary. In terms of the command line, the CLI has moved to a separate package, called `webpack-cli`. You need to install it before using Webpack 4.

When updating plugins, many third-party plugins need to be upgraded to their latest version to be compatible, so please be aware of that. It is also a good idea to peruse your project to find the ones that need updating. Also, be sure to add the new mode option to your configuration:

1. Begin by setting the mode in your configuration to `production` or `development`, depending on the configuration type, as in the following code snippet, using `webpack.config.js`:

   ```
   module.exports = {
       mode: 'production',
   }
   ```

   There is an alternative method, which can be done by passing the mode using the CLI, such as with the following example:

   ```
   --mode production
   ```

   The preceding example shows the latter part of any Webpack command for the `production` mode made through the command line. The following example shows the same for the `development` mode:

   ```
   --mode development
   ```

2. The next step is the removal of deprecated plugins; the plugins should be removed from your configuration file as they are default in production mode. The following example will show you how to make the edit in `webpack.config.js`:

   ```
   module.exports = {
    plugins: [
      new NoEmitOnErrorsPlugin(),
      new ModuleConcatenationPlugin(),
      new DefinePlugin({ "process.env.NODE_ENV":
        JSON.stringify("production") })
      new UglifyJsPlugin()
    ],
   }
   ```

   The following example gives you a view of how this works in development mode. Note that the plugins are the default in development mode, again using `webpack.config.js`:

   ```
   module.exports = {
    plugins: [
      new NamedModulesPlugin()
    ],
   }
   ```

3. If that was done correctly, you will see that the depreciated plugins have been removed. Your configuration file, `webpack.config.js`, should look something like the following:

   ```
   module.exports = {
    plugins: [
      new NoErrorsPlugin(),
      new NewWatchingPlugin()
    ],
   }
   ```

Also, `CommonChunkPlugin` was removed in this process with the `optimization.splitChunks` options offered as an alternative in Webpack 4.0.

If you are generating HTML from stats, `optimization.splitChunks.chunks: "all"` can now be used—this is the optimal configuration in most cases.

There is also some work to be done regarding `import()` and CommonJS. When using `import()` to load any non-ESM scripts, the result has changed in Webpack 4:

1. Now, you need to access the default property to get the value of `module.exports`. See the `non-esm.js` file here to see this in action:

   ```
   module.exports = {
         sayHello: () => {
          console.log('Hello World');
         }
    };
   ```

   This is a simple JavaScipt function and you can replicate its contents to follow the demonstration and see how the results have changed.

2. The next file is an `example.js` file. It can be called anything you want and you can perform any action you want. In this example, it is a simple `sayHello();` function:

   ```
   function sayHello() {
    import('./non-esm.js').then(module => {
     module.default.sayHello();
    });
   }
   ```

   These blocks show how to code simple functions with CommonJS. You should apply this convention to your existing code to ensure it doesn't break.

3. When using a custom loader to transform `.json` files, you now need to change the module type in `webpack.config.js`:

   ```
   module.exports = {
    rules: [
     {
       test: /config\.json$/,
       loader: 'special-loader',
       type: 'javascript/auto',
       options: {...}
     }
    ]
   };
   ```

4. Even when using `json-loader`, it can be removed; see the following example:

```
module.exports = {
  rules: [
   {
     test: /\.json$/,
     loader: 'json-loader'
   }
  ]
};
```

Once this is done, all of the required migration prerequisites will have been done. The next step is the automated updating process, which is built into Webpack.

Prerequisites when migrating to version 5 from version 4

This guide aims to help you migrate to Webpack 5 when using Webpack directly. If you are using a higher-level tool to run Webpack, please refer to this tool for migration instructions.

As explained in `Chapter 1`, *Introduction to Webpack 5*, Webpack 5 requires Node.js version 10.13.0 (LTS) to run; however, the use of a newer version can improve build performance even more:

1. You should be sure to check related plugins and loaders for individual migration instructions through the author-provided copies, especially when upgrading across major versions. In such cases, be aware of deprecation warnings during the build. You can invoke Webpack this way to get stack traces for deprecation warnings to figure out which plugins and loaders are responsible. Webpack 5 will remove all the deprecated features. To proceed, there should be no deprecation warnings during the build.
2. Make sure you are using entry point information from stats. If you are using `HtmlWebpackPlugin`, you won't need to follow this step.

   For builds that include static HTML or the creation thereof in some other way, you must ensure the use of entry points for the stats JSON file to generate any script and link any HTML tags. If this isn't possible, you should avoid setting the `splitChunks.chunks` key to `all` and putting any setting against the `splitChunks.maxSize` key. However, this is merely a workaround and can be considered less than ideal as a solution.

3. Be sure to set the mode to either `production` or `development` to make sure that corresponding mode defaults are set.
4. Also, be sure to update the following options to their newer version if you are using them at all:

```
optimization.hashedModuleIds: true => optimization.moduleIds:
  'hashed'
optimization.namedChunks: true => optimization.chunkIds: 'named'
optimization.namedModules: true => optimization.moduleIds: 'named'
NamedModulesPlugin => optimization.moduleIds: 'named'
NamedChunksPlugin => optimization.chunkIds: 'named'
HashedModulesPlugin => optimization.moduleIds: 'hashed'
optimization.occurrenceOrder: true => optimization: { chunkIds:
   'total-size', moduleIds: 'size' }
optimization.splitChunks.cacheGroups.vendors =>
   optimization.splitChunks.cacheGroups.defaultVendors
```

5. Next, we will need to test Webpack 5's compatibility with your application. To do this, set the following options for your Webpack 4 configuration. If this works in Webpack 4 without any build errors, we will know whether any successive faults are unique to version 5. This may sound tedious, but it eliminates recursive fault-finding:

```
module.exports = {
 // ...
   node: {
     Buffer: false,
     process: false
   }
 };
```

The preceding options are removed from the configuration in Webpack 5 and are set to `false` by default. Be sure to do this in your Webpack 4 test build, but they will need removing again in your version 5 build.

6. Next is a simple and shorthand command-line execution to upgrade your Webpack version:

```
npm: npm install webpack@next --dev
Yarn: yarn add webpack@next -D
```

Now, we need to clean up our configuration.

It is advised that you change the `[hash]` placeholder in your configuration to `[contenthash]`. This has been proven to be more effective and can help to shore up your code.

If you happen to be using `pnp-webpack-plugin`, it is now supported by default in version 5 of Webpack but it will now need to be removed from your configuration schema.

`IgnorePlugin` now takes an option object, so it will need to be rewritten if you are using it as a regular expression, such as in the following:

```
new IgnorePlugin({ resourceRegExp: /regExp/ }).
```

For developers using WASM via `import`, you should enable the deprecated specification by setting the `experiments.syncWebAssembly` variable to `true`. This will set the same behavior in Webpack 5 as in Webpack 4. Once you have migrated to Webpack 5, you should now change the value of the experiment to use the latest specifications for WASM—`{ asyncWebAssembly: true, importAsync: true }`.

You should also take care when using a custom configuration to replace the `name` value with `idHint`.

In Webpack 5, named exports from JSON modules are not supported and you will get a warning. To import anything this way, you should do so from `package.json` with `const[version]=package;`.

7. It is now good practice to clean up the build code. Part of this means closing the compiler when using `const compiler =webpack(...);`. This is done with `compiler.close();`.

Once you run a build, there may be a few issues that develop. For instance, the schema validation could fail, the Webpack could exit with an error, or there could be build errors, build warnings, or deprecation warnings.

In each case, there will be either a breaking change note or an error message with instructions available through the command line, as usual.

In the case of deprecation warnings, there may be a lot of them for the time being as Webpack 5 is new and plugins need time to catch up with core changes. They should be ignored until each release is out of beta testing as a matter of good practice.

You can hide deprecation warnings by running the node with the `--no-deprecation` flag—for example, `node --no-deprecation`.

Plugin and loader contributors should follow the warning advice in the deprecation messages to improve their code.

You may also want to turn off the ES2015 syntax in the runtime code, if necessary. By default, Webpack's runtime code uses ES2015 syntax to build smaller bundles. If your build targets environments that don't support this syntax, such as with IE 11, you will need to set `output.ecmaVersion: 5` to revert to ES5 syntax.

Dealing with legacy issues will be the biggest hurdle when migrating upwards and this rule is not exclusive to Webpack 5. Webpack 5 has some features that will make the experience of the legacy platform user more palatable. One method to consider in your project planning is persistent caching.

Enabling persistent caching

Caching is, of course, the intermediate storage of data to improve loading times and speed up performance. Persistent caching is something that is very common in database-driven projects, where the data pulled from the database is cached so the user has a copy of earlier versions. This can then be loaded all at once without causing too much demand from the database as data will be delivered at a slower rate than server-based file entries.

With Webpack 5, an application can utilize the same operation and improve the speed of loading for the user if the build changes.

First, note that persistent caching is not enabled by default. You have to opt in to using it. This is because Webpack 5 favors safety over performance. It is probably not the best idea to enable features that improve performance by even a small amount but break your code in any small way. At least as a default, this feature should remain disabled.

Serialization and deserialization would work by default; however, the developer may have trouble with cache invalidation.

Cache invalidation is when something changes in your application intentionally, such as when the developer changes the contents of a file; in this case, Webpack would regard the caching of the old content as invalid.

Webpack 5 does this by tracking `fileDependencies`, `contextDependencies`, and `missingDependencies` for each module used. Webpack then creates a filesystem schematic from this. The filesystem is then cross-referenced against the recorded copy and this, in turn, triggers a rebuild of that module.

The cache entry of the output bundle then has a tag generated for it; this is essentially a hash of all contributors. A match between the tag and the cache entry indicates content that can be used by Webpack for bundling.

Webpack 4 used this same process for in-memory caching and it will work in Webpack 5 without extra configuration unless persistent caching is enabled.

You also need to invalidate cache entries when you upgrade a loader or plugin with `npm`, change a configuration, or change a file that is to be read in the configuration, or when upgrading a dependency that is used in the configuration, when passing difference command-line arguments to run a build, or when you have a custom build script and make changes to it.

As Webpack 5 cannot handle these exceptions out of the box, persistent caching is made an opt-in feature for safety in regards to securing the integrity of your application.

Updating Webpack

There are numerous steps that must be taken to ensure the update of Webpack behaves correctly. The steps that concern our example are as follows:

- Upgrade and install.
- Add the modes to your configuration files.
- Add fork checkers.
- Manually update relevant plugins, loaders, and utilities.
- Reconfigure `uglify`.
- Trace any further errors and make the update.

Let's go over each step in detail and explore what exactly is going on with the command line. It should help you to understand the procedure a lot better:

1. The first thing we need to do is upgrade Webpack and install `webpack-cli`. This is done in the command line, as follows:

```
yarn add webpack
yarn add webpack-cli
```

2. The preceding example shows how this is done using `yarn`; it will also give a version check. This should also be visible in the `package.json` file:

   ```
   ...
   "webpack": "^5.0.0",
   "webpack-cli": "^3.2.3",
   ...
   ```

3. Once this is done, the respective modes should be added to `webpack.config.dev.js` and `webpack.config.prod.js`. See the following `webpack.config.dev.js` file:

   ```
   module.exports = {
   mode: 'development',
   ```

   A similar thing is done with the production configuration, as we have two configuration files here for each mode. The following shows the contents of the `webpack.config.prod.js` file:

   ```
   module.exports = {
   mode: 'production',
   ```

   We are dealing with two versions—the older version (3) and the newer version (4). If this was done manually, you might first make a **fork** of the original version. The term fork refers to the icon usually associated with this operation, which represents one line splitting away from the other to appear as a two-pronged fork. So, the term fork has come to mean a subversion. Fork checkers will automatically check each version for differences that need to be updated as part of the operation.

4. Now, go back to the command line to add the following fork checkers:

   ```
   add fork-ts-checker-notifier-webpack-plugin
   yarn add fork-ts-checker-notifier-webpack-plugin --dev
   ```

   The following should be seen in the `package.json` file:

   ```
   ...
   "fork-ts-checker-notifier-webpack-plugin": "^1.0.0",
   ...
   ```

   The preceding code block shows that the fork checker has been installed.

5. Now, we need to update `html-webpack-plugin` with the command line:

   ```
   yarn add html-webpack-plugin@next
   ```

   `package.json` should now show the following:

   ```
   "html-webpack-plugin": "^4.0.0-beta.5",
   ```

   Now, we need to adjust the plugin order in the `webpack.config.dev.js` and `webpack.config.prod.js` files.

6. You should take these steps to ensure that `HtmlWebpackPlugin` comes before `InterpolateHtmlPlugin` and `InterpolateHtmlPlugin` are declared, as in the following example:

   ```
   plugins: [
    new HtmlWebpackPlugin({
      inject: true,
      template: paths.appHtml
    }),
    new InterpolateHtmlPlugin(HtmlWebpackPlugin, env.raw),
   ```

7. Also, be sure to update `ts-loader`, `url-loader`, and `file-loader` in the command line:

   ```
   yarn add url-loader file-loader ts-loader
   ```

8. The `package.json` file holds information on the versions used, in terms of the previously mentioned loaders, and should look as in the following:

   ```
   "file-loader": "^1.1.11",
   "ts-loader": "4.0.0",
   "url-loader": "0.6.2",
   ```

   If you're using **React**, then you will need to update the development utilities, as follows:

   ```
   yarn add react-dev-utils
   ```

   Again, the `package.json` file will hold the version information for the React utility in use:

   ```
   "react-dev-utils": "6.1.1",
   ```

`extract-text-webpack-plugin` should be substituted with `mini-css-extract-plugin`.

9. Take note that `extract-text-webpack-plugin` should be removed altogether while adding and configuring `mini-css-extract-plugin`:

```
yarn add mini-css-extract-plugin
```

10. The `package.json` file with version settings for the plugin should look as in the following for this example:

```
"mini-css-extract-plugin": "^0.5.0",
Config:
```

11. With all this done, we should then take a look at the production mode configurations. This is done with the following `webpack.config.prod.js` file:

```
const MiniCssExtractPlugin = require("mini-css-extract-plugin");
plugins: [
 ...
 new MiniCssExtractPlugin({
   filename: "[name].css",
   chunkFilename: "[id].css"
 }),
 ...
 ],
 module: {
   rules: [
    {
     test: /\.css$/,
     use: [
     {
       loader: MiniCssExtractPlugin.loader,
       options: {
       // you can specify a publicPath here
       // by default it use publicPath in webpackOptions.output
       publicPath: '../'
     }
    },
    "css-loader"
   ]
 },
```

We can see some differences in `webpack.config.prod.js` between versions. The preceding gives you a look at the format for configuration when in version 4.

12. Next, be sure to update and reconfigure `uglifyjs-webpack-plugin` using the command line and the `package.json` file:

```
yarn add uglifyjs-webpack-plugin --dev
```

13. In the interests of prudence, we will also show the version settings for the `uglify` plugin here. Apply these using `package.json`:

```
"uglifyjs-webpack-plugin": "^2.1.2"
 Config:
```

14. The next and final step is the configuration for production mode using `webpack.config.prod.js`:

```
const UglifyJsPlugin = require('uglifyjs-webpack-plugin');
module.exports = {
  ...
  optimization: {
    minimizer: [new UglifyJsPlugin()],
 },
```

Once this is all done, you should be done with the updating process. However, you may get a unique deprecation error, which means you will need to trace these errors using the error message then update any further Webpack plugins as required. This will especially be the case if you are working with custom plugins or loaders.

Summary

This chapter took a deep look into code debugging and we discussed HMR and other debugging techniques. You should now have a firm grasp of what tools and utilities can be used to enhance debugging procedures, including the use of `node nightly` to perform code inspections. We then delved into HMR, a salient and exciting feature of Webpack. We saw how live edits can be made of modules and style sheets, and even covered issues with migration. We then segued into adding utilities, which is an essential part of any upgrade. From there, we took you through version migration—namely, from version 3 to version 4—and the steps to follow to do so. Furthermore, we showed you how to migrate from version 4 to version 5. This section concluded with a lengthy tutorial on updating a command-line upgrade to a manual alteration of some of the trickier elements.

You should now be confident in your debugging and upgrading skills, which puts you on solid ground for the next chapter. In the next chapter, we will get into some heavy live coding, customization, and manual bundling, which will excite the pants off you, no doubt!

Further reading

This chapter covered some complex issues that could be better understood through further reading. The following is a list of subjects and where to find the corresponding content alluded to earlier in this chapter, where relevant:

- Debugging optimization bailouts
- Issue 6074—add support for more complex selectors for `sideEffects`

Questions

Now, try your hand at some of the following questions related to this chapter. You will find the answers in the *Assessment* section in the back matter of this book:

1. What does HMR stand for?
2. What does React Hot Loader do?
3. What interface is Webpack updated through?
4. What feature of Node v6.3.0+ allows debugging via Chrome DevTools?
5. When migrating from Webpack version 3 to version 4 and using a custom loader to transform `.json` files, what must you also change?
6. How can a side effects list help development?
7. Where should deprecated plugins be removed from in production mode?

8
Authoring Tutorials and Live Coding Hacks

This is the final chapter of this book for Webpack 5.0. By now, you may feel like an expert, but proof of your mastery comes both from your ability to take ownership of the code itself and to customize the platform, and even to hack into it. A purist might shy away from terms such as *hacking* in favor of workarounds or patching, but we are essentially talking about the same subject (except explaining to the layperson that you have hacked Webpack during live production would certainly impress the most ardent skeptic of your coding superpowers).

That being said, this chapter is a compilation of the hardest things to do in Webpack and the easiest ways to accomplish them. We will begin with authoring libraries and move on to custom loaders, something discussed in this guide already, particularly with the use of Babel's **application programming interface** (**API**). This chapter will, however, discuss a more native approach to customization without the need for an API.

We will, naturally, cover some topics already discussed in some detail in this guide, such as testing and shimming, but this guide will offer a little more salience and relevance to customization.

From there, we will cover some very interesting hacks, specifically when working with live publications in a **Hot Module Replacement** (**HMR**)-enabled environment.

Some of the topics covered in this final chapter are as follows:

- Authoring libraries
- Custom loaders
- Live coding hacks

Authoring libraries

This section of this chapter will be of great use to anyone hoping to streamline their bundling strategy. It is not very well known that Webpack can be used for bundling both libraries and applications.

We will begin with a hypothetical custom library project, which we'll call `numbers-to-text`. It will work by converting numerals from, say, `1` to `5` to a textual representation of the number, such as `3` to `three`.

Let's discuss each task in detail and explain what is happening in the code, with examples to help us understand with greater clarity exactly what is happening and how the code behaves. We will proceed as follows:

1. We will begin by getting our project structure in line. The basic project structure should look like this:

   ```
   |- webpack.config.js
     |- package.json
     |- /src
     |- index.js
     |- ref.json
   ```

   Note that the structure may look different from the previous tutorial—namely, the presence of the `ref.json` file. You will need to create those extra files if you don't have them before we go any further.

2. Next, we come back to the **command-line interface** (**CLI**), and we first need to initialize `npm`, then make sure we have installed Webpack and `lodash`. If you have already installed this by following the example in previous chapters, then don't worry—duplicate installation attempts will only overwrite the last and shouldn't cause any harm at all. Run the following code:

   ```
   npm init -y
   npm install --save-dev webpack lodash
   ```

3. That being done, we avert our attention to the newly built `src/ref.json` **JSON** file. This is the essential data of the custom library. It should look like the following example:

   ```
   [
     {
       "num": 1,
       "word": "One"
     },
   ```

```
  {
    "num": 2,
    "word": "Two"
  },
  {
    "num": 3,
    "word": "Three"
  },
  {
    "num": 4,
    "word": "Four"
  },
  {
    "num": 5,
    "word": "Five"
  },
  {
    "num": 0,
    "word": "Zero"
  }
]
```

As you can see, this constitutes a simple list of options representing a number, with a corresponding written word version of that number. This will form the backbone of our very simple library structure, to demonstrate the concept in principle. Once the tutorial is complete, you should naturally see how you can adapt the library to your needs, no matter how complex.

4. Now, we need to make an index file (such as `src/index.js`). You should follow the coding displayed in this block:

```
import _ from 'lodash';
import numRef from './ref.json';
export function numToWord(num) {
  return _.reduce(numRef, (accum, ref) => {
  return ref.num === num ? ref.word : accum;
 }, '');
}
export function wordToNum(word) {
  return _.reduce(numRef, (accum, ref) => {
  return ref.word === word && word.toLowerCase() ? ref.num : accum;
  }, -1);
}
```

You can see from the preceding code how the index file essentially contains a series of `export` and `return` functions related to the **JSON** file contents.

5. Now, we need to define the usage specification. This is done as follows:

    - ES2015 module import:

    ```
     import * as numbersToText from 'numbers-to-text';
    // ...
     numbersToText.wordToNum('Two');
    ```

    - CommonJS module require:

    ```
     const numbersToText = require('numbersToText');
    // ...
    ```

6. Use the following snippet of code, in the same file, to set the functions when using `AMD`:

    ```
    numbersToText.wordToNum('Two');
    AMD module requires:
    require(['numbersToText'], function (numbersToText) {
     numbersToText.wordToNum('Two');
     });
    ```

    The user can also use the library by loading it via a `script` tag, like this:

    ```
    <!doctype html>
     <html>
       ...
       <script src="https://unpkg.com/webpack-numbers"></script>
       <script>
        // Global variable
        numbersToText.wordToNum('Five')
        // Property in the window object
        window.numbersToText.wordToNum('Five')
       </script>
     </html>
    ```

This is the most common way in which a library is loaded, and, as a JavaScript developer, it's something that should be second nature to follow. That being said, it can be configured to expose a property in the global object for Node.js or as a property in a `this` object.

This takes us to the basic configuration for the library. There is more than one level of configuration for authorizing this library, so this is just the first step.

The basic configuration

As with any Webpack project, we need to configure it. This requires some extra attention when dealing with a custom library. Several unusual things must be achieved with this configuration as opposed to a typical one. Let's consider these objectives now. The library should be bundled in a way that achieves the following goals:

- The use of externals to avoid bundling `lodash` so the user is required to load it
- Specifying the external limitations of the library
- Exposing the library as a variable called `numbersToText`
- Setting the library name to `numbers-to-text`
- Permitting the access to the **Node.js** library indie

Also, note that the user must be able to access and have use of the library in the following ways:

- Through importing `numbersToText` from `numbers-to-text` as an **ECMAScript 2015** (**ES 2015**) module
- Through the CommonJS module, such as using the `require('webpack-numbers')` method
- Through a global variable when included through such methods as a script tag

With all that in mind, the first thing to do is set up the Webpack configuration through the normal files we use for this `webpack.config.js` file, as follows:

```
const path = require('path');
module.exports = {
  entry: './src/index.js',
  output: {
   path: path.resolve(__dirname, 'dist'),
   filename: 'numbers-to-text.js'
  }
};
```

That is the basic configuration, but we'll now move on to the next goal identified: the externalizing of `lodash`.

Using externals to avoid bundling lodash

If you now perform a build, you will see that quite a large bundle is created. Inspecting the file reveals that `lodash` has been bundled alongside it. For the sake of this tutorial, `lodash` is better treated as a peer dependency. This essentially means that the user would have `lodash` installed, effectively giving control of this external library to the user of this library.

This can be done through the configuration of externals, as in `webpack.config.js`, as follows:

```
const path = require('path');
module.exports = {
  entry: './src/index.js',
  output: {
   path: path.resolve(__dirname, 'dist'),
   filename: 'numbers-to-text.js'
   }
  },
   externals: {
     lodash: {
      commonjs: 'lodash',
      commonjs2: 'lodash',
      amd: 'lodash',
      root: '_'
    }
  }
};
```

The preceding code means that the library expects a dependency named `lodash` to be available in the user's environment.

Note that externals may be specified as an array if the plan is to use the library as a dependency in a parallel Webpack bundle.

It's also important to specify a limitation on the externals, and we will discuss this now.

Specifying external limitations

You may work with libraries that use several files from a dependency, such as in the following descriptive block:

- `import A from 'library/one';`
- `import B from 'library/two';`

In this case, they can't be excluded from the bundle by specifying the library in the externals. They would need to be excluded one at a time or by using a regular expression, *s*uch as with the following example:

```
module.exports = {
 externals: [
   'library/one',
   'library/two',
   // Everything that starts with "library/"
   /^library\/.+$/
  ]
};
```

Once that is accomplished, we need to expose the library or allow it to be loaded into our frontend. This is covered in the next subsection.

Exposing the library

Exposing a library is something already discussed in this guide, but if you're jumping into this chapter, you may be puzzled. We are simply allowing our application to load the library from an external source, as with any library externally loading into a web page, for example. The library should be compatible with different environments, such as **CommonJS**, **MD**, or **Node.js** to ensure the widespread usability of the library. To ensure this, proceed as follows:

1. The `library` property should be added inside the output in the `webpack.config.js` configuration file, as follows:

```
const path = require('path');
module.exports = {
  entry: './src/index.js',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'numbers-to-text.js'
    filename: 'numbers-to-text.js',
    library: 'numbersToText'
```

```
 },
 externals: {
   lodash: {
    commonjs: 'lodash',
    commonjs2: 'lodash',
    amd: 'lodash',
    root: '_'
  }
 }
};
```

The library setup is related to configuration. In most cases, specifying one entry point is enough. Multiple-part libraries are possible; however, it is simpler to expose partial exports through an index script, serving as the entry point.

TIP It is unwise to attempt to use an array to a library entry point, as it won't compile very well.

This achieves the exposure of the library bundle as a global variable.

2. You can make your library compatible with other environments by adding a `libraryTarget` property to the configuration file, adding different options on how they expose the library, using `webpack.config.js`, as follows:

```
const path = require('path');
module.exports = {
  entry: './src/index.js',
  output: {
   path: path.resolve(__dirname, 'dist'),
   filename: 'numbers-to-text.js',
   library: 'numbersToText'
   library: 'numbersToText',
   libraryTarget: 'umd'
 },
 externals: {
   lodash: {
    commonjs: 'lodash',
    commonjs2: 'lodash',
    amd: 'lodash',
    root: '_'
   }
  }
};
```

With the configuration set, we now need to expose the library. Note that this can be done in the following ways:

- As a variable—as a global variable made available by a script tag, such as `libraryTarget:'var'`
- As an object—available through a `this` object, such as `libraryTarget:'this'`
- Window—this is available through the `window` object, such as `libraryTarget:'window'`
- **Universal Module Definition** (**UMD**)—available after CommonJS or AMD `require` statements, such as `libraryTarget:'umd'`

If you set the library but don't do this for the `libraryTarget` function, the latter will default to a variable, as specified in the output configuration.

Naming the library and working with Node.js

As explained, we are now in the last steps of this authoring. The optimization of the output should be made by following the guide as we proceed. While we do this, we will add in the path to the bundle output as the package's main field with the `package.json` file, as follows:

```
{
 ...
 "main": "dist/numbers-to-text.js",
 ...
 }
 Or, to add as standard module as per this guide:
{
 ...
 "module": "src/index.js",
 ...
 }
```

The option key named `"main"` in the preceding code block refers to the standard we retrieve from the `package.json` file. The `"module"` key refers to a proposal that allows the JavaScript environment to upgrade, to be able to use ES2015 modules without harming any backward-compatibility capabilities.

In the case of the `"module"` property, this should point to a script that uses ES2015 module syntax throughout, but no other syntax that is not yet supported by browsers or Node.js. This will enable Webpack to parse module syntax and allow lighter bundles through tree shaking, as users are likely only consuming certain parts of any given library.

Once this is done, the bundle can be published as an `npm` package.

You have learned how to set up and configure your first custom library using a JSON file with corresponding numbers and text, including exposing the new library in your frontend and specifying the limitations in scope for the library.

In a related sphere, let's now talk about custom loaders.

Custom loaders

Loaders have been discussed in detail in previous chapters of this guide. However, we have only alluded to the customization or authoring of them. This will be increasingly important to at least demonstrate your mastery of Webpack, so we should discuss it now.

The following tutorial will be structured like this:

- Setup
- Simple usage
- Complex usage
- Guidelines

The *Guidelines* section will itself be subcategorized, but for now, let's begin with the setup.

Setup

The best way to start this section is to look at how we can develop and test a loader locally. This is a nice and palatable way to begin, and we will proceed as follows:

1. When testing a single loader, you can simply use a path to resolve to a local file within a rule object in `webpack.config.js`, as follows:

```
module.exports = {
 //...
 module: {
 rules: [
 {
   test: /\.js$/,
   use: [
 {
    loader: path.resolve('path/to/loader.js'),
    options: {/* ... */}
    }
```

```
        ]
      }
    ]
  }
};
```

2. To test multiple loaders, you can utilize the `resolveLoader.modules` configuration, whereby Webpack will search for loaders in `webpack.config.js`. For example, if you had a local directory in your project with a loader inside of it, the code would look like this:

```
module.exports = {
  resolveLoader: {
    modules: [
     'node_modules',
     path.resolve(__dirname, 'loaders')
    ]
  }
};
```

That should be all you need to begin. However, if you've already created a separate repository for your loader, you could use an `npm` link to the project in which you'd like to run the test.

There's more than one methodology depending on the usage of the loader, so—naturally—we will begin with simple usage.

Simple usage

The idea of simple loaders has already been alluded to, this being that a loader is more useful when it performs a very simple and specific task. This will make testing easier and, as there are so many, they can be *chained* to others in a more complex usage to perform a greater variety of tasks.

When a single loader is applied to the resource, the loader is called with only one parameter. This is a string containing the content of the resource being loaded.

Synchronous loaders can return a single value representing the transformed module. In more complex cases, the loader can return any number of values by using the following function: `this.callback(err, values...)`.

Errors are then either passed to the function or thrown in a synchronized loader.

In this case, the loader is expected to give back one or two values. The first value is some resulting JavaScript code as a string. The second value is optional and results in a `SourceMap` and JavaScript object.

Loaders tend to get more complex in situations where they are chained. When discussing complex usage of custom loaders, that would be a good place to start, so let's do that now.

Complex usage

As discussed in the previous subsection, *Simple usage*, complex usage generally refers to the use of a loader in context with another or a set of loaders in a chained pattern.

When multiple loaders are chained, it is important to remember that they are executed in reverse order!

This will be either right to left or bottom to top, depending on the array format you use. For instance, the following will apply:

- The last loader, which is called by the script first, will be passed the contents of the raw resource (the data or script being run by the loader).
- The first loader, called last, is expected to return JavaScript and an optional source map.
- The loaders in between will be executed with the result(s) of the previous loader in the chain.

So, in the following common example, `foo-loader` would be passed the raw resource, and `bar-loader` would receive the output of `foo-loader` and return the final transformed module and a source map, if necessary.

To chain loaders this way, start with the `webpack.config.js` configuration file, like this:

```
module.exports = {
  //...
  module: {
    rules: [
      {
        test: /\.js/,
        use: [
         'bar-loader',
         'foo-loader'
        ]
```

```
          }
        ]
      }
    };
```

That's the configurations, but as chained loaders are complex by nature, it's good to follow a standard. What follows is a set of guidelines that will help your project come together without fraying at the edges.

Guidelines

The following guidelines should be followed when writing a loader. They are ordered in terms of importance, and some only apply in certain scenarios:

- Simplifying the purpose of the loader
- Utilizing chaining
- Modular outputting
- Ensuring statelessness
- Employing loader utilities
- Marking loader dependencies
- Resolving module dependencies
- Extracting common code
- Avoiding absolute paths at all costs!
- Using peer dependencies

Let's go through each one in more detail, starting with simplification.

Simplifying the purpose of the loader

Loaders work best when they perform a simple and clear task. This can make the job of maintaining each loader simpler, and also permits chaining for use in more complex tasks. This is because there may be many different loaders specific to the different tasks; so, to allow versatility, they are often used in sequence.

Therefore, much as with Webpack bundles, they should be modular. Therefore, the specific task they perform can be isolated and refined, which will also allow more ubiquitous application when used in a chain with other loaders. This brings us to the next concept: the chaining of a loader.

Utilizing chaining

Take advantage of the fact that loaders can be chained together. Instead of writing a single loader that tackles many tasks, write multiple loaders. Isolating them not only keeps each loader simple but may also allow them to be used for something more varied.

When rendering a template file with data specified via loader options or query parameters, for example, it could be written as a single loader that compiles the template from source, executes it, and returns a module that exports a string containing the HTML code. However, in the following guidelines, a simple `apply-loader` exists that can be chained with other open source loaders:

- `jade-loader`: This converts the template to a module that exports a function.
- `apply-loader`: This executes the function with loader options and returns basic HTML code.
- `e idea HTML-loader`: This accepts the HTML code and outputs a valid JavaScript module.
- The fact that loaders can be chained also means they don't necessarily have to output JavaScript, as long as the next loader in the chain can handle its output.

Webpack is always modular, so let's look at the advice concerning that when working with chained loaders.

Modular outputting

You should always, and in the best of circumstances, keep output modular. Loader-generated modules should adhere to the same design heuristics as normal modules.

This may be for obvious reasons, but compatibility with existing projects would mean that this standard should be followed. It will also be of increasing importance for the chaining of loaders, as discussed previously. Loaders are often used in sequence with each other, and many Webpack projects require the installation and usage of many of them.

Therefore, the adherence to a modular output convention will prevent projects from becoming overly complicated and, in fact, possibly cause a reversal in the purpose of Webpack bundling, which is making a smaller, more succinct, and optimized application.

This conventional adherence or standard formatting is something that will be second nature to most developers, but when working with Webpack, there may be some compatibility considerations you have overlooked as they are so peculiar to bundling. One of these considerations is the "state" of the loader.

Ensuring statelessness

Make sure the loader does not retain a state between module transformations. Each run should always be independent of other compiled modules.

During the compilation process, you may end up running several builds as you fine-tune your bundle, and you don't want to correct each run, but rather retain each build in its original state.

This will make things much easier to keep track of should things go wrong, as you can always start again with the source files without starting at the very beginning of your command-line session.

It is also important to consider compatibility with other loaders. As this is a convention among loaders, you should maintain that same convention unless it is fundamentally necessary for your loader to perform its specific task, and, if so, this should be made clear to the developer so that no mistakes are made.

If the passing of a state is necessary for the functionality of your loader, then there is a convenient solution to provide adherence to conventions: the loader utilities package.

Employing loader utilities

Why not take advantage of the `loader-utils` package? It provides a variety of useful tools, but one of the most common of these is the ability to retrieve the options being passed to any loader in use. Along with `loader-utils`, the `schema-utils` package should be used for consistent JSON Schema-based validation. The following code block shows an example that utilizes both packages, using `loader.js`:

```
import { getOptions } from 'loader-utils';
 import validateOptions from 'schema-utils';
const schema = {
  type: 'object',
  properties: {
    test: {
     type: 'string'
    }
  }
 };
export default function(source) {
  const options = getOptions(this);
  validateOptions(schema, options, 'Example Loader');
```

Now, we can apply some transformations to the source, like this:

```
return `export default ${ JSON.stringify(source) }`;
 }
```

This transformation will **stringify** the code—essentially, outputting the contents into a single line of code that is difficult to read by humans but ideal for computers. This also often helps with privacy issues, should anyone wish to manually copy the code. With that understood, let's move on to loader dependency guidelines.

Marking loader dependencies

If a loader uses external resources, such as when reading from a filesystem, the loader must indicate this. This information is used to invalidate "cacheable" loaders and recompile them in watch mode. What follows is a brief example of how to accomplish this, using the `addDependency` method inside of `loader.js`:

```
import path from 'path';
export default function(source) {
  var callback = this.async();
  var headerPath = path.resolve('header.js');
  this.addDependency(headerPath);
  fs.readFile(headerPath, 'utf-8', function(err, header) {
    if(err) return callback(err);
    callback(null, header + '\n' + source);
  });
 }
```

There are some differences between the loader and modular dependencies. Let's now discuss the latter.

Resolving module dependencies

Depending on the type of module you are using, there may be a different schema in use to specify any dependency. In **Cascading Style Sheets** (**CSS**), for example, the `@import` and `URL(...)` statements are used. When doing this, these dependencies should be resolved by the module system.

This can be achieved in one of the following two ways:

- By transforming the statements to `require` statements
- Using the `this.resolve` function to resolve a path

The `css-loader` is a good example of the first approach. It transforms dependencies to `require` statements, by replacing `@import` statements with a request to the other style sheet and `url(...)` with a request to the file being referenced.

In respect to the LESS loader, each `@import` statement is unable to transform to a `require` statement because **fewer** files must be compiled in a single iteration. As a result, the LESS loader will use custom path-resolving logic to extend the LESS compiler. It will then use the `this.resolve` method to resolve the dependency.

If you're using language that only accepts relative **Uniform Resource Locators** (**URLs**), the ~ tilde convention can be used to specify references to the installing modules. An example would be `url('~some-library/image.png').`

Extracting common code

As part of best practice here, the generating of common code in every module of the loader process should be avoided. A better approach is to use a runtime file in the loader and generate a `require` statement process to any shared module. This better suits how Webpack parses code.

It is the essential purpose of Webpack to compile a project so that code is not duplicated, so this may go without saying, but the loaders themselves should do this and not leave it to the Webpack core processing. Otherwise, applications would be needlessly large or have needlessly long compilation times.

If you have experience in programming plugins, you may overlook this very obvious precept, but it is worth mentioning here as it is so essential to the operation and processes of Webpack.

Avoiding absolute paths

As alluded to, inserting absolute paths into any code related to a module should not be done, as hashing will break if the root directory is ever moved. Also, note that there is a `stringifyRequest` method in the `loader-utils` loader, which can be used to an absolute path to a relative one to help automate your process.

Refer to `Chapter 2`, *Working with Modules and Code Splitting,* to get a refresher on absolute paths if you think you need it.

As with common code, this is something very fundamental to how Webpack works, and if you don't have that in mind during your authoring you may overlook it, so it is certainly worth mentioning. Relative paths are the way to go.

One last point on standards relates to peer dependencies. Let's look at these now.

Using peer dependencies

In the event of developing a loader that is a simple wrapper (essentially, code that acts as a shell for more operational code within), the operational code—or package—should be included as `peerDependency`. This is because it will allow you to specify the exact version of the package using the `package.json` file.

In the following example, the `sass-loader` specifies `node-sass` as a peer dependency. Take a look at the code:

```
{
 "peerDependencies": {
   "node-sass": "^4.0.0"
  }
}
```

This can be invaluable for compatibility issues, especially on complex programming projects.

Unit testing

Thus far, we have written a custom loader, followed the guidelines, and even got it running locally. The next step is testing. The following example is a simple unit testing procedure. It makes use of the `babel-jest` **Jest** framework and some other presets to allow the use of the `import/export` and `async/await` methods:

1. We'll start by installing and saving these as something called `devDependencies`, like this:

   ```
   npm install --save-dev jest babel-jest babel-preset-env
   ```

   The previous command-line entry installs the **Jest** framework and **Babel** with **Jest** in development mode.

2. Next, we must look at the configuration used in `webpack.config.js` concerning this particular unit testing procedure, as follows:

   ```
   .babelrc
   {
    "presets": [[
    "env",
    {
    "targets": {
    "node": "4"
    }
    }
    ]]
    }
   ```

3. The function of the loader in the example is to process a text file and replace any instance of `[name]` with the option given to the loader. It then outputs a valid **JavaScript** module containing the text as its default export, as seen in the following example inside of `src/loader.js`:

   ```
   import { getOptions } from 'loader-utils';
   export default function loader(source) {
    const options = getOptions(this);
   source = source.replace(/\[name\]/g, options.name);
   return `export default ${ JSON.stringify(source) }`;
    }
   ```

4. This loader will be used to process the following text file, called `test/example.txt`:

   ```
   Hi Reader!
   ```

5. The next step is a little more complicated. It uses the **Node.js** API and `memory-fs` to execute Webpack. This will avoid content being output to the local hard drive (very handy to know) and gives us access to statistical data that can be used to take hold of our transformed module. It begins with the following command line:

   ```
   npm install --save-dev webpack memory-fs
   ```

6. Once it's installed, there's some work we need to do on its associated compiler script. Use the following `test/compiler.js` file:

   ```
   import path from 'path';
    import webpack from 'webpack';
   ```

```
 import memoryfs from 'memory-fs';
export default (fixture, options = {}) => {
 const compiler = webpack({
  context: __dirname,
  entry: `./${fixture}`,
  output: {
   path: path.resolve(__dirname),
   filename: 'bundle.js',
 },
 module: {
  rules: [{
  test: /\.txt$/,
  use: {
   loader: path.resolve(__dirname, '../src/loader.js'),
   options: {
     name: 'Alice'
    }
   }
 }]
 }
});
compiler.outputFileSystem = new memoryfs();
return new Promise((resolve, reject) => {
  compiler.run((err, stats) => {
 if (err) reject(err);
 if (stats.hasErrors()) reject(new Error(stats.toJson().errors));
resolve(stats);
 });
});
};
```

In the previous example, we inlined our configuration, but a configuration as a parameter to the `export` function can also be accepted. This allows the testing of multiple setups using the same compiler module.

7. This being done, we can now write our test and add an `npm` script to run it. Let's begin by adding the following code to our `test/loader.test.js` file:

```
import compiler from './compiler.js';
test('Inserts name and outputs JavaScript', async () => {
 const stats = await compiler('example.txt');
 const output = stats.toJson().modules[0].source;
expect(output).toBe('export default "Hi Reader!\\n"');
 });
 package.json
{
 "scripts": {
```

```
 "test": "jest"
}
}
```

The preceding code block shows the testing function that loads in the example text, should the program work correctly.

Everything should now be in place.

8. The code can now be run, and we'll check whether the new loader has passed the test by running an `npm` build in the command line and viewing the command-line window, as follows:

```
 Inserts name and outputs JavaScript (229ms)
Test Suites: 1 passed, 1 total
Tests: 1 passed, 1 total
Snapshots: 0 total
Time: 1.853s, estimated 2s
Ran all test suites.
```

If you see something like the preceding text in the corresponding file, then it passed. Well done!

At this point, you should be able to develop, test, and deploy your loaders. Don't forget to share your creations with the rest of the Webpack community and help expand the capabilities of developers everywhere, and make your mark at the same time.

We've covered a huge amount in this guide and I suppose you feel like an expert, having built custom libraries and loaders, but making live coding hacks would make your skillset even more impressive. These are useful to know, especially on custom jobs where workarounds and makeshift approaches are more likely than tried-and-tested code, so let's dive right in!

Live coding hacks

In this section, we will get to see some very intriguing stuff that will leave an expert feeling like a superhero, should he or she ever get into hot water or simply wish to show off. We will discuss loaders that work well with **HMR**, such as `monkey-hot-loader` and `react-hot-loader`, as well as the various uses of `eval` and `__Eval`.

To begin with, you should note that there is a side effect with **HMR**. It always evaluates the entire module again when it updates. This includes the dependency chain, which updates to point to new modules. However, we may just want the original module to evaluate the new code, not the entire module. Thankfully, there is a hack around this: using `monkey-hot-loader`!

Monkey Hot Loader

This will also mean that if your module has side effects, such as starting a server, then `monkey-hot-loader` won't work that well with it. That won't be the case if there is a global state, but there really shouldn't be one if it's coded correctly.

Also, note that when we change a module by taking `monkey-patch` and patch the original module with the updates.

In this section, we will explore how patching top-level functions works in detail.

`monkey-hot-loader` is a Webpack loader that parses the JavaScript file and extracts all the names of the top-level functions in the file:

1. For example, take a look at the following code, which we are placing in the `app.js` file but can be placed anywhere, as the code works globally and will affect top-level functions:

   ```
   function foo() {
     return 5;
   }
   function bar() {
     return function() {
     // ...
    }
   }
   module.exports = function() {
    // ...
   }
   ```

   In respect to the previous example, the latest version of `monkey-hot-loader` currently only extracts the function names `foo` and `bar`, as only these functions can be patched. A few other types of functions could be made patchable, but let's keep things simple for now for the sake of explanation.

The `foo` and `bar` functions can be patched by setting them to new functions. This is because they are top-level functions. When updated, the functions will be created in the same scope. It's then easy to inject new code there.

There is only one real problem with that, concerning exporting the function: modules using the exported function would still be referencing the old variation. Quite a bit of work needs to be done to get around that, which we will go over now.

The names of these functions are given to the runtime code that `monkey-hot-loader` appends to each module.

2. When the module runs initially, it will iterate over these names and make each function patchable, and we do this by replacing it with the following code:

```
var patched = function() {
 if(patchedBindings[binding]) {
  return patchedBindings[binding].apply(this, arguments);
 }
 else {
  return f.apply(this, arguments);
 }
};
 patched.prototype = f.prototype;
```

Here, the `f` variable would reference the name `foo` if we were patching it. Note that the semantics of `patched` should be the same as the `f` variable. Any call to the `patched` variable should produce the same result as a call to `f`.

With the initial semantics of `foo` intact, we have installed a "hook" to perform a check to see whether there's a new version of the function to call. After all top-level functions are replaced with this variation, we can simply override any of them by loading a function into `patchedBindings`. Even exported functions will call the new variation.

Currently, the `monkey-hot-loader` implements this top-level function patching as an initial experiment. You may consider playing with it by using the `backend-with-webpack` project to see how to integrate it with your application.

Depending on the context, a different heuristic may need to be adopted for patching. For instance, if your frontend uses **React**, most of your code will exist inside the **React** library components. The `react-hot-loader` works fantastically well for that. However, concerning your backend code, most of it might be classed with methods, in which case patching the method on the prototype works best for that.

React Hot Patching

`react-hot-loader` works by binding the original module to any new code, whether it be functions, classes, or methods. It will patch all of the methods of **React** components to use the new methods.

The question this leaves is how to patch the original module. Things will get extremely complicated if you try to accept an update. For instance, if you change code inside a closure, it becomes hard to patch the closure without losing the existing state. It may be achievable to use the **React** engine's debugger API but to make this change throughout may be difficult: how difficult will depend greatly on your specific context.

Note that when patching closures, only allowing very basic patching is best as it's intuitively easy to keep track of how everything works.

We often have to patch code when it's a complex and custom project. One tool that is very common with Webpack is an innate patching tool called `eval`. We will take a closer look at that now.

Eval

Whether it's React Hot Patching or Monkey Hot Loader, we can install all these patch versions throughout the module's scope using `eval`. After that, we save the scope of the module. This will only happen when the module runs for the first time:

1. If we need the ability to alter code in this specific scope, we can do that by creating an `eval` proxy to maintain any state in the `app.js` file, as follows:

   ```
   var moduleEval = function(some code) {
    return eval(some code);
    }
   ```

   This function is later passed to future variations of this module through the dispose handler.

   While all of the aforementioned happened on the initial run of the module in question, consecutive updates through a different route. In this case, the entire module is evaluated again except iterating over each top-level binding, then a call is made to `func.toString()` to return the function code, then the code is reevaluated in the scope of the original module using `moduleEval`—to reference the original state.

2. Then, this `eval` function is installed in `patchedBindings` so that it is used in any future calls made by the system, as follows:

```
bindings.forEach(function(binding) {
 // Get the updated function instance
 var f = eval(binding);
// We need to rectify the function in the original module so
 // it references any of the original state. Strip the name
 // and simply eval it.
 var funcCode = (
 '(' + f.toString().replace(/^function \w+\(/, 'function (') + ')'
 );
 patchedBindings[binding] = module.hot.data.moduleEval(funcCode);
 });
```

In an ideal situation, we might get the source code of the module updated and avoid running the module since we just want the code in the form of a string anyway.

As it happens, it is possible to avoid the whole `func.toString()` and `moduleEval` process and simply not support any global state, although the global state is very useful for debugging operations. This is especially true for interactions that are simple to **REPL** (**Read–Eval–Print Loop**). Classes, however, don't have this issue, insomuch that all of their states are part of the instance, which is why `react-hot-loader` works just fine without this hack.

For the uninitiated, a REPL is also known as an interactive top-level or language shell, which takes single user inputs and evaluates them.

Be aware that in Webpack 5, there are currently known problems with `eval()` that relate to `optimization.innerGraph` when in production mode.

There is a hack available for `_eval`, and it's very useful to know. Let's get to that part now.

The __Eval hack

Time for one last hack, and it is probably the biggest trick in this whole book! The `_eval()` function evaluates a string as an expression of a function. This can be used in conjunction with hot loading to allow an immediate evaluation of your code right across your project. This is essentially what the `_Eval` hack is. Let's explore it a little more now.

If we want a REPL that evaluates code inside a module and to be able to open modules to choose which context to evaluate in, then we can't do that with this infrastructure, but we can get most of the way there with the following example in the `app.js` file:

```
function __eval() {
 var user = getLastUser();
 console.log(findAllDataOn(user));
 }
```

Once this is done and you define a function named `__eval`, `monkey-hot-loader` will execute it every time the module updates. This is highly useful for instant feedback. With this approach, you could call some APIs and log the results, then work on those APIs on the fly until you see the results you want. That way, all you have to do is make some coding alterations, save the file, and then instantly see the updated output.

Also, you can use the code form `__eval` as a globally used script and allow the typical **HMR** system to run the module every time it's updated. This being said, any module with side effects will need specialist code. What's more, you can build a state across evaluations to play with or debug.

Unlike the old **Lisp** style of doing things (that is, select code and press *Ctrl* + *E* to run), this technique is done per module, and you have the latitude to select a context to run the code in.

One consideration is that new variables cannot be introduced, such as changing a variable's value, using the `_eval` function in `strict` mode. This will also apply to `__eval`, so it's worth bearing in mind before you begin.

Summary

This chapter has taken you through some of the more advanced features that are possible with Webpack, such as library authoring and live coding hacks, using the `_Eval` technique with hot loading to allow instant feedback across your project. This has included detailed explanations and examples of how to customize loaders, and even patch top-level functions.

You should now have an understanding of manual bundling and live coding, deep enough to parallel any expert. Why not demonstrate this know-how to yourself by taking the quiz at the very end of this chapter? It should stand you in good stead if you are ever put on the spot at job interviews or even presentations to major clients, to be able to quickly express your expertise.

This book as a whole has given extensive and comprehensive detail on both how to use Webpack competently and take your application development to completely new levels. This particular chapter will be increasingly important as your Webpack bundling develops, and you can be sure that this chapter specifically will be one bookmarked for many projects to come.

Once you have tried the practice quiz questions, you may want to flip back to the start of the book and test yourself on each chapter. You will find the assessment answers in a separate chapter at the back of this guide. Successful completion will result in your expertise being beyond doubt, so give it a try.

Questions

1. Can Webpack be used to bundle libraries as well as applications?
2. How are external libraries excluded from the bundle when authoring libraries?
3. Webpack offers four ways to expose a custom library. What are they?
4. Why shouldn't absolute paths be used when building a custom module?
5. How can the function prefix of `__eval` help with instant feedback?
6. Why must a developer indicate the reading of external resources such as a filesystem by a loader?
7. How are loaders executed when they are chained?

Assessment

Answers

This section comprises the answers to the questions given to the reader at the end of each chapter. The reader should refer to this section when testing themselves or others.

It is recommended to first read each chapter carefully or as many times as you feel you will need to fully digest the information.

After this, you may want to get a piece of scrap paper and jot down your answers to each set of questions before turning to this *Assessment* section and checking to see if you are right. You may want to award yourself a mark for each correct answer and see if you can improve on that score later.

Good luck!

Chapter 1: Introduction to Webpack 5

1. Webpack is a module bundler for JavaScript applications.
2. The bundle is the output file generated by Webpack. It contains all of the modules which are used in the application. The bundle generation process is regulated by Webpack config file.
3. 4.29.6 or 4.0 is acceptable.
4. Node.js
5. Any time one file depends on another, Webpack treats this as a dependency.
6. `install`
7. NPM
8. With the entry of a minus sign in front of the `lodash` call.
9. The "distribution" code is the minimized and optimized output of our build process that will eventually be loaded in the browser.
10. To ensure we keep our package as private, also removing the main entry. By doing this we can prevent accidentally publishing your code.

Chapter 2: Working with Modules and Code Splitting

1. Code splitting is the process of automatically organizing your programming ins a modular format. Modular programming is the wider concept.
2. A Chunk refers to a group of modules.
3. Dynamic imports are essentially on-demand imports into Webpack. An entry point is a fixed and configured Entry Pint where the build begins processing code from.
4. **Module Chunks** using the preload directive loads in parallel to its parent 'chunk', whereas a prefetched chunk starts after the parent chunk finishes loading. **Chunks** must be instantly requested by the parent chunk when preloaded, whereas a prefetched can be used at any time. **Chunks** using the preload directive are instantly downloaded when called. A prefetched chunk is downloaded while the browser is idle.
5. Code linting is the process of removing the unwanted or surplus code.
6. A promise refers to information returned form a loader.
7. The **`SplitChunksPlugin`** allows the extraction of common dependencies into entry chunks.
8. This tool will analyze your bundle and give suggestions to reduce your bundle size.
9. This offers an Interactive pie chart used for Webpack stats.
10. It is a method for displaying hierarchical data using nested figures, usually rectangles.

Chapter 3: Using Configurations and Options

1. The configuration is done via a set of Configuration Files and Options are set using Command-Line.
2. A command-line technique which informs the bundler, what configuration file to use.
3. The file-loader.
4. JSON files.

5. Each file that Webpack generates.
6. This will force Webpack to exit its bundling process.
7. This option will limit the number of parallel processed modules.
8. It will specify the file from which the last set of records is read
9. It will disable **AMD** support.
10. Compiling is the process by which Webpack 5 assembles the information including assets

Chapter 4: APIs Loaders and Plugins

1. Internationalization (i18n) is the process of preparing software so that it supports local languages and cultural settings
2. Babel
3. To convert **ECMAScript** to an earlier version for compatibility purposes.
4. Loader-builder
5. The loader allows you to write mixed HTML, CSS and JavaScript Polymer element
6. The ability to make mixed-use HTML, CSS and JavaScript files and process them in the Webpack environment.
7. The compiler.

Chapter 5: Libraries and Frameworks

1. Vue's template compiler.
2. `Main.ts` and `Vendor.ts`
3. **Node v6.9.0** installed and **Webpack v4.0.0.**
4. To inspect them for the command-line interface.
5. A Single-page application.
6. Webpack's configuration file.
7. The `@` symbol: `import '@angular/http';`
8. A JavaScript tag in the HTML file.
9. An open-source package manager that is similar to NPM, that comes part of Node.JS.
10. Runtime only **ECMAScript** module compilation

Chapter 6: Deployment and Installation

1. When using languages that are considered verbose, the programmer must write a lot of code to accomplish only minor functionality. Such code is called boilerplate.
2. Tree shaking is a term for dead-code elimination.
3. Essentially to polyfill or patch code.
4. To deliver a native application experience online.
5. Handle the automation of code.
6. Gulp, Mocha and Karma.
7. ECMAScript 2015, CommonJS and AMD.

Chapter 7: Debugging and Migration

1. Hot Module Replacement.
2. It tweaks React components in real-time.
3. Command Line
4. Inspection flag.
5. The module type.
6. By preventing the unintentional dropping of code.
7. The configuration file, `webpack.config.js`.

Chapter 8: Authoring Tutorials and Live Coding

1. Yes.
2. One by one or by using a regular expression.
3. As a variable, an object, through a window or using **UMD**.
4. It can cause a hashing break when the root for the project is moved
5. As it will execute it every single time the module is updated
6. Because this information is used to invalidate cacheable loaders and recompile in watch mode.
7. In reverse order, either right to left or bottom to top depending on array format.

Leave a review - let other readers know what you think

Index