Table of Contents

Introduction 1.1

CONCEPTS 1.2

- Concepts 1.2.1
- Entry Points 1.2.2
- Output 1.2.3
- Mode 1.2.4
- Loaders 1.2.5
- Plugins 1.2.6
- Configuration 1.2.7
- Modules 1.2.8
- Module Resolution 1.2.9
- Dependency Graph 1.2.10
- The Manifest 1.2.11
- Targets 1.2.12
- Hot Module Replacement 1.2.13

CONFIGURATION 1.3

- Configuration 1.3.1
- Configuration Languages 1.3.2
- Configuration Types 1.3.3
- Entry and Context 1.3.4
- Output 1.3.5
- Module 1.3.6
- Resolve 1.3.7
- Plugins 1.3.8
- DevServer 1.3.9
- Devtool 1.3.10
- Target 1.3.11
- Watch and WatchOptions 1.3.12
- Externals 1.3.13
- Performance 1.3.14
- Node 1.3.15
- Stats 1.3.16
- Other Options 1.3.17

API 1.4

- Introduction 1.4.1
- Command Line Interface 1.4.2
- Stats Data 1.4.3

- Node.js API 1.4.4
- Hot Module Replacement 1.4.5
- Loader API 1.4.6
- MODULES 1.4.7
  - Module Methods 1.4.7.1
  - Module Variables 1.4.7.2
- PLUGINS 1.4.8
  - Introduction 1.4.8.1
  - Introduction 1.4.8.2
  - Introduction 1.4.8.3
  - Introduction 1.4.8.4
  - Introduction 1.4.8.5

GUIDES 1.5

- Guides 1.5.1
- Installation 1.5.2
- Getting Started 1.5.3
- Asset Management 1.5.4
- Output Management 1.5.5
- Development 1.5.6
- Hot Module Replacement 1.5.7
- Tree Shaking 1.5.8
- Production 1.5.9
- Code Splitting 1.5.10
- Lazy Loading 1.5.11
- Caching 1.5.12
- Authoring Libraries 1.5.13
- Shimming 1.5.14
- Progressive Web Application 1.5.15
- TypeScript 1.5.16
- Migrating Versions 1.5.17
- Environment Variables 1.5.18
- Build Performance 1.5.19
- Content Security Policies 1.5.20
- Development Vagrant 1.5.21
- Dependency Management 1.5.22
- Public Path 1.5.23
- Integration 1.5.24

LOADERS 1.6

- Loaders 1.6.1

PLUGINS 1.7

Plugins	1.7.1
AggressiveSplittingPlugin	1.7.2
BannerPlugin	1.7.3
CommonsChunkPlugin	1.7.4
ContextReplacementPlugin	1.7.5
DefinePlugin	1.7.6
DllPlugin	1.7.7
EnvironmentPlugin	1.7.8
EvalSourceMapDevToolPlugin	1.7.9
HashedModuleIdsPlugin	1.7.10
HotModuleReplacementPlugin	1.7.11
HtmlWebpackPlugin	1.7.12
LimitChunkCountPlugin	1.7.13
LoaderOptionsPlugin	1.7.14
MinChunkSizePlugin	1.7.15
ModuleConcatentationPlugin	1.7.16
NamedModulePlugin	1.7.17
NoEmitOnErrorsPlugin	1.7.18
NormalModulesReplacementPlugin	1.7.19
PerfectPlugin	1.7.20
ProfillingPlugin	1.7.21
ProvidePlugin	1.7.22
SourceMapDevToolPlugin	1.7.23
splitChunksPlugin	1.7.24
WatchIgnorePlugin	1.7.25
IgnorePlugin	1.7.26

webpack.js.org

Guides, documentation, and all things webpack.

Content Progress

Now that we've covered much of the backlog of *missing documentation*, we are starting to heavily review each section of the site's content to sort and structure it appropriately. The following issues should provide a pretty good idea of where things are, and where they are going:

- Guides - Review and Simplify
- Concepts - Review and Organize
- API - v4 Rewrite

We haven't created issues for the other sections yet, but they will be coming soon. For dev-related work please see General - Updates & Fixes.

Translation

To help translate this documentation please jump to the translate branch.

Contributing

Read through the writer's guide if you're interested in editing the content on this site. See the contributors page to learn how to set up and start working on the site locally.

License

Special Thanks

BrowserStack has graciously allowed us to do cross-browser and cross-os testing of the site at no cost...

At its core, *webpack* is a *static module bundler* for modern JavaScript applications. When webpack processes your application, it recursively builds a *dependency graph* that includes every module your application needs, then packages all of those modules into one or more *bundles*.

T> Learn more about JavaScript modules and webpack modules here.

Since v4.0.0 webpack does not require a configuration file. Nevertheless, it is incredibly configurable. To get started you only need to understand four **Core Concepts**:

- Entry
- Output
- Loaders
- Plugins

This document is intended to give a **high-level** overview of these concepts, while providing links to detailed concept specific use cases.

Entry

An **entry point** indicates which module webpack should use to begin building out its internal *dependency graph*. After entering the entry point, webpack will figure out which other modules and libraries that entry point depends on (directly and indirectly).

Every dependency is then processed and outputted into files called *bundles*, which we'll discuss more in the next section.

You can specify an entry point (or multiple entry points) by configuring the `entry` property in the webpack configuration. It defaults to `./src`.

Here's the simplest example of an `entry` configuration:

webpack.config.js

```
module.exports = {
  entry: './path/to/my/entry/file.js'
};
```

T> You can configure the `entry` property in various ways depending the needs of your application. Learn more in the entry points section.

Output

The **output** property tells webpack where to emit the *bundles* it creates and how to name these files, it defaults to `./dist`. Basically, the entire app structure will get compiled into the folder that you specify in the output path. You can configure this part of the process by specifying an `output` field in your configuration:

webpack.config.js

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

module.exports = {
  entry: './path/to/my/entry/file.js',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'my-first-webpack.bundle.js'
  }
```

```
};
```

In the example above, we use the `output.filename` and the `output.path` properties to tell webpack the name of our bundle and where we want it to be emitted to. In case you're wondering about the path module being imported at the top, it is a core Node.js module that gets used to manipulate file paths.

T> You may see the term **emitted** or **emit** used throughout our documentation and plugin API. This is a fancy term for 'produced' or 'discharged'.

T> The `output` property has many more configurable features and if you like to know more about the concepts behind the `output` property, you can read more in the concepts section.

Loaders

Loaders enable webpack to process more than just JavaScript files (webpack itself only understands JavaScript). They give you the ability to leverage webpack's bundling capabilities for all kinds of files by converting them to valid modules that webpack can process.

Essentially, webpack loaders transform all types of files into modules that can be included in your application's dependency graph (and eventually a bundle).

W> Note that the ability to `import` any type of module, e.g. `.css` files, is a feature specific to webpack and may not be supported by other bundlers or task runners. We feel this extension of the language is warranted as it allows developers to build a more accurate dependency graph.

At a high level, **loaders** have two purposes in your webpack configuration:

1. The `test` property identifies which file or files should be transformed.
2. The `use` property indicates which loader should be used to do the transforming.

webpack.config.js

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

const config = {
  output: {
    filename: 'my-first-webpack.bundle.js'
  },
  module: {
    rules: [
      { test: /\.txt$/, use: 'raw-loader' }
    ]
  }
};

module.exports = config;
```

The configuration above has defined a `rules` property for a single module with two required properties: `test` and `use`. This tells webpack's compiler the following:

> "Hey webpack compiler, when you come across a path that resolves to a '.txt' file inside of a `require()` / `import` statement, **use** the `raw-loader` to transform it before you add it to the bundle."

W> It is important to remember that **when defining rules in your webpack config, you are defining them under `module.rules` and not `rules`**. For your benefit, webpack will 'yell at you' if this is done incorrectly.

There are other, more specific properties to define on loaders that we haven't yet covered.

Learn more!

Plugins

While loaders are used to transform certain types of modules, plugins can be leveraged to perform a wider range of tasks. Plugins range from bundle optimization and minification all the way to defining environment-like variables. The plugin interface is extremely powerful and can be used to tackle a wide variety of tasks.

In order to use a plugin, you need to `require()` it and add it to the `plugins` array. Most plugins are customizable through options. Since you can use a plugin multiple times in a config for different purposes, you need to create an instance of it by calling it with the `new` operator.

webpack.config.js

```
const HtmlWebpackPlugin = require('html-webpack-plugin'); //installed via npm
const webpack = require('webpack'); //to access built-in plugins

const config = {
  module: {
    rules: [
      { test: /\.txt$/, use: 'raw-loader' }
    ]
  },
  plugins: [
    new webpack.optimize.UglifyJsPlugin(),
    new HtmlWebpackPlugin({template: './src/index.html'})
  ]
};

module.exports = config;
```

There are many plugins that webpack provides out of the box! Check out our list of plugins for more information.

Using plugins in your webpack config is straightforward - however, there are many use cases that are worth further exploration.

Learn more!

Mode

By setting the `mode` parameter to either `development` or `production`, you can enable webpack's built-in optimizations that correspond with the selected mode.

```
module.exports = {
  mode: 'production'
};
```

Learn more!

As mentioned in Getting Started, there are multiple ways to define the `entry` property in your webpack configuration. We will show you the ways you **can** configure the `entry` property, in addition to explaining why it may be useful to you.

Single Entry (Shorthand) Syntax

Usage: `entry: string|Array<string>`

webpack.config.js

```
const config = {
  entry: './path/to/my/entry/file.js'
};

module.exports = config;
```

The single entry syntax for the `entry` property is a shorthand for:

```
const config = {
  entry: {
    main: './path/to/my/entry/file.js'
  }
};
```

T> **What happens when you pass an array to `entry` ?** Passing an array of file paths to the `entry` property creates what is known as a **"multi-main entry"**. This is useful when you would like to inject multiple dependent files together and graph their dependencies into one "chunk".

This is a great choice when you are looking to quickly setup a webpack configuration for an application or tool with one entry point (IE: a library). However, there is not much flexibility in extending or scaling your configuration with this syntax.

Object Syntax

Usage: `entry: {[entryChunkName: string]: string|Array<string>}`

webpack.config.js

```
const config = {
  entry: {
    app: './src/app.js',
    vendors: './src/vendors.js'
  }
};
```

The object syntax is more verbose. However, this is the most scalable way of defining entry/entries in your application.

T> **"Scalable webpack configurations"** are ones that can be reused and combined with other partial configurations. This is a popular technique used to separate concerns by environment, build target and runtime. They are then merged using specialized tools like webpack-merge.

Scenarios

Below is a list of entry configurations and their real-world use cases:

Separate App and Vendor Entries

webpack.config.js

```
const config = {
  entry: {
    app: './src/app.js',
    vendors: './src/vendors.js'
  }
};
```

What does this do? At face value this tells webpack to create dependency graphs starting at both `app.js` and `vendors.js`. These graphs are completely separate and independent of each other (there will be a webpack bootstrap in each bundle). This is commonly seen with single page applications which have only one entry point (excluding vendors).

Why? This setup allows you to leverage `CommonsChunkPlugin` and extract any vendor references from your app bundle into your vendor bundle, replacing them with `__webpack_require__()` calls. If there is no vendor code in your application bundle, then you can achieve a common pattern in webpack known as long-term vendor-caching.

?> Consider removing this scenario in favor of the DllPlugin, which provides a better vendor-splitting.

Multi Page Application

webpack.config.js

```
const config = {
  entry: {
    pageOne: './src/pageOne/index.js',
    pageTwo: './src/pageTwo/index.js',
    pageThree: './src/pageThree/index.js'
  }
};
```

What does this do? We are telling webpack that we would like 3 separate dependency graphs (like the above example).

Why? In a multi-page application, the server is going to fetch a new HTML document for you. The page reloads this new document and assets are redownloaded. However, this gives us the unique opportunity to do multiple things:

- Use `CommonsChunkPlugin` to create bundles of shared application code between each page. Multi-page applications that reuse a lot of code/modules between entry points can greatly benefit from these techniques, as the amount of entry points increase.

T> As a rule of thumb: for each HTML document use exactly one entry point.

Configuring the `output` configuration options tells webpack how to write the compiled files to disk. Note that, while there can be multiple `entry` points, only one `output` configuration is specified.

Usage

The minimum requirements for the `output` property in your webpack config is to set its value to an object including the following two things:

- A `filename` to use for the output file(s).
- An absolute `path` to your preferred output directory.

webpack.config.js

```
const config = {
  output: {
    filename: 'bundle.js',
    path: '/home/proj/public/assets'
  }
};

module.exports = config;
```

This configuration would output a single `bundle.js` file into the `/home/proj/public/assets` directory.

Multiple Entry Points

If your configuration creates more than a single "chunk" (as with multiple entry points or when using plugins like CommonsChunkPlugin), you should use substitutions to ensure that each file has a unique name.

```
{
  entry: {
    app: './src/app.js',
    search: './src/search.js'
  },
  output: {
    filename: '[name].js',
    path: __dirname + '/dist'
  }
}

// writes to disk: ./dist/app.js, ./dist/search.js
```

Advanced

Here's a more complicated example of using a CDN and hashes for assets:

config.js

```
output: {
  path: "/home/proj/cdn/assets/[hash]",
  publicPath: "http://cdn.example.com/assets/[hash]/"
}
```

In cases when the eventual `publicPath` of output files isn't known at compile time, it can be left blank and set dynamically at runtime in the entry point file. If you don't know the `publicPath` while compiling, you can omit it and set `__webpack_public_path__` on your entry point.

```
__webpack_public_path__ = myRuntimePublicPath

// rest of your application entry
```

Providing the `mode` configuration option tells webpack to use its built-in optimizations accordingly.

`string`

Usage

Just provide the `mode` option in the config:

```
module.exports = {
  mode: 'production'
};
```

or pass it as a CLI argument:

```
webpack --mode=production
```

The following string values are supported:

Option	Description
`development`	Provides `process.env.NODE_ENV` with value `development`. Enables `NamedChunksPlugin` and `NamedModulesPlugin`.
`production`	Provides `process.env.NODE_ENV` with value `production`. Enables `FlagDependencyUsagePlugin`, `FlagIncludedChunksPlugin`, `ModuleConcatenationPlugin`, `NoEmitOnErrorsPlugin`, `OccurrenceOrderPlugin`, `SideEffectsFlagPlugin` and `UglifyJsPlugin`.

T> Please remember that setting `NODE_ENV` doesn't automatically set `mode`.

Mode: development

```
// webpack.development.config.js
module.exports = {
+ mode: 'development'
- plugins: [
-   new webpack.NamedModulesPlugin(),
-   new webpack.DefinePlugin({ "process.env.NODE_ENV": JSON.stringify("development") }),
- ]
}
```

Mode: production

```
// webpack.production.config.js
module.exports = {
+  mode: 'production',
-  plugins: [
-    new UglifyJsPlugin(/* ... */),
-    new webpack.DefinePlugin({ "process.env.NODE_ENV": JSON.stringify("production") }),
-    new webpack.optimize.ModuleConcatenationPlugin(),
-    new webpack.NoEmitOnErrorsPlugin()
-  ]
}
```

Loaders are transformations that are applied on the source code of a module. They allow you to pre-process files as you `import` or “load” them. Thus, loaders are kind of like “tasks” in other build tools, and provide a powerful way to handle front-end build steps. Loaders can transform files from a different language (like TypeScript) to JavaScript, or inline images as data URLs. Loaders even allow you to do things like `import` CSS files directly from your JavaScript modules!

Example

For example, you can use loaders to tell webpack to load a CSS file or to convert TypeScript to JavaScript. To do this, you would start by installing the loaders you need:

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

And then instruct webpack to use the `css-loader` for every `.css` file and the `ts-loader` for all `.ts` files:

webpack.config.js

```
module.exports = {
  module: {
    rules: [
      { test: /\.css$/, use: 'css-loader' },
      { test: /\.ts$/, use: 'ts-loader' }
    ]
  }
};
```

Using Loaders

There are three ways to use loaders in your application:

- Configuration (recommended): Specify them in your **webpack.config.js** file.
- Inline: Specify them explicitly in each `import` statement.
- CLI: Specify them within a shell command.

Configuration

`module.rules` allows you to specify several loaders within your webpack configuration. This is a concise way to display loaders, and helps to maintain clean code. It also offers you a full overview of each respective loader:

```
  module: {
    rules: [
      {
        test: /\.css$/,
        use: [
          { loader: ['style-loader'](/loaders/style-loader) },
          {
            loader: ['css-loader'](/loaders/css-loader),
            options: {
              modules: true
            }
          }
        ]
      }
    ]
  }
```

Inline

It's possible to specify loaders in an `import` statement, or any equivalent "importing" method. Separate loaders from the resource with `!`. Each part is resolved relative to the current directory.

```
import Styles from 'style-loader!css-loader?modules!./styles.css';
```

It's possible to overwrite any loaders in the configuration by prefixing the entire rule with `!`.

Options can be passed with a query parameter, e.g. `?key=value&foo=bar`, or a JSON object, e.g. `?{"key":"value","foo":"bar"}`.

T> Use `module.rules` whenever possible, as this will reduce boilerplate in your source code and allow you to debug or locate a loader faster if something goes south.

CLI

You can also use loaders through the CLI:

```
webpack --module-bind jade-loader --module-bind 'css=style-loader!css-loader'
```

This uses the `jade-loader` for `.jade` files, and the `style-loader` and `css-loader` for `.css` files.

Loader Features

- Loaders can be chained. They are applied in a pipeline to the resource. A chain of loaders are executed in reverse order. The first loader in a chain of loaders returns a value to the next. At the end loader, webpack expects JavaScript to be returned.
- Loaders can be synchronous or asynchronous.
- Loaders run in Node.js and can do everything that's possible there.
- Loaders accept query parameters. This can be used to pass configuration to the loader.
- Loaders can also be configured with an `options` object.
- Normal modules can export a loader in addition to the normal `main` via `package.json` with the `loader` field.
- Plugins can give loaders more features.
- Loaders can emit additional arbitrary files.

Loaders allow more power in the JavaScript ecosystem through preprocessing functions (loaders). Users now have more flexibility to include fine-grained logic such as compression, packaging, language translations and more.

Resolving Loaders

Loaders follow the standard module resolution. In most cases it will be loaders from the module path (think `npm install`, `node_modules`).

A loader module is expected to export a function and be written in Node.js compatible JavaScript. They are most commonly managed with npm, but you can also have custom loaders as files within your application. By convention, loaders are usually named `xxx-loader` (e.g. `json-loader`). See "How to Write a Loader?" for more information.

Plugins are the backbone of webpack. webpack itself is built on the **same plugin system** that you use in your webpack configuration!

They also serve the purpose of doing **anything else** that a loader cannot do.

Anatomy

A webpack **plugin** is a JavaScript object that has an `apply` property. This `apply` property is called by the webpack compiler, giving access to the **entire** compilation lifecycle.

ConsoleLogOnBuildWebpackPlugin.js

```
const pluginName = 'ConsoleLogOnBuildWebpackPlugin';

class ConsoleLogOnBuildWebpackPlugin {
    apply(compiler) {
        compiler.hooks.run.tap(pluginName, compilation => {
            console.log("The webpack build process is starting!!!");
        });
    }
}
```

First parameter of the tap method of the compiler hook should be a camelized version of the plugin name. It is advisable to use a constant for this so it can be reused in all hooks.

Usage

Since **plugins** can take arguments/options, you must pass a `new` instance to the `plugins` property in your webpack configuration.

Depending on how you are using webpack, there are multiple ways to use plugins.

Configuration

webpack.config.js

```
const HtmlWebpackPlugin = require('html-webpack-plugin'); //installed via npm
const webpack = require('webpack'); //to access built-in plugins
const path = require('path');

const config = {
  entry: './path/to/my/entry/file.js',
  output: {
    filename: 'my-first-webpack.bundle.js',
    path: path.resolve(__dirname, 'dist')
  },
  module: {
    rules: [
      {
        test: /\.(js|jsx)$/,
        use: 'babel-loader'
      }
    ]
  },
  plugins: [
    new webpack.optimize.UglifyJsPlugin(),
    new HtmlWebpackPlugin({template: './src/index.html'})
  ]
};
```

```
module.exports = config;
```

Node API

?> Even when using the Node API, users should pass plugins via the `plugins` property in the configuration. Using `compiler.apply` should not be the recommended way.

some-node-script.js

```
const webpack = require('webpack'); //to access webpack runtime
const configuration = require('./webpack.config.js');

let compiler = webpack(configuration);
compiler.apply(new webpack.ProgressPlugin());

compiler.run(function(err, stats) {
  // ...
});
```

T> Did you know: The example seen above is extremely similar to the webpack runtime itself! There are lots of great usage examples hiding in the webpack source code that you can apply to your own configurations and scripts!

You may have noticed that few webpack configurations look exactly alike. This is because **webpack's configuration file is a JavaScript file that exports an object.** This object is then processed by webpack based upon its defined properties.

Because it's a standard Node.js CommonJS module, you **can do the following**:

- import other files via `require(...)`
- use utilities on npm via `require(...)`
- use JavaScript control flow expressions i. e. the `?:` operator
- use constants or variables for often used values
- write and execute functions to generate a part of the configuration

Use these features when appropriate.

While they are technically feasible, **the following practices should be avoided**:

- Access CLI arguments, when using the webpack CLI (instead write your own CLI, or use `--env`)
- Export non-deterministic values (calling webpack twice should result in the same output files)
- Write long configurations (instead split the configuration into multiple files)

T> The most important part to take away from this document is that there are many different ways to format and style your webpack configuration. The key is to stick with something consistent that you and your team can understand and maintain.

The following examples below describe how webpack's configuration object can be both expressive and configurable because *it is code*:

Simple Configuration

webpack.config.js

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

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

Multiple Targets

See: Exporting multiple configurations

Using other Configuration Languages

webpack accepts configuration files written in multiple programming and data languages.

See: Configuration Languages

In modular programming, developers break programs up into discrete chunks of functionality called a *module*.

Each module has a smaller surface area than a full program, making verification, debugging, and testing trivial. Well-written *modules* provide solid abstractions and encapsulation boundaries, so that each module has a coherent design and a clear purpose within the overall application.

Node.js has supported modular programming almost since its inception. On the web, however, support for *modules* has been slow to arrive. Multiple tools exist that support modular JavaScript on the web, with a variety of benefits and limitations. webpack builds on lessons learned from these systems and applies the concept of *modules* to any file in your project.

What is a webpack Module

In contrast to Node.js modules, webpack *modules* can express their *dependencies* in a variety of ways. A few examples are:

- An ES2015 `import` statement
- A CommonJS `require()` statement
- An AMD `define` and `require` statement
- An `@import` statement inside of a css/sass/less file.
- An image url in a stylesheet (`url(...)`) or html (`<img src=...>`) file.

T> webpack 1 requires a specific loader to convert ES2015 `import` , however this is possible out of the box via webpack 2

Supported Module Types

webpack supports modules written in a variety of languages and preprocessors, via *loaders*. *Loaders* describe to webpack **how** to process non-JavaScript *modules* and include these *dependencies* into your *bundles*. The webpack community has built *loaders* for a wide variety of popular languages and language processors, including:

- CoffeeScript
- TypeScript
- ESNext (Babel)
- Sass
- Less
- Stylus

And many others! Overall, webpack provides a powerful and rich API for customization that allows one to use webpack for **any stack**, while staying **non-opinionated** about your development, testing, and production workflows.

For a full list, see **the list of loaders** or **write your own**.

A resolver is a library which helps in locating a module by its absolute path. A module can be required as a dependency from another module as:

```
import foo from 'path/to/module'
// or
require('path/to/module')
```

The dependency module can be from the application code or a third party library. The resolver helps webpack find the module code that needs to be included in the bundle for every such `require` / `import` statement. webpack uses enhanced-resolve to resolve file paths while bundling modules.

Resolving rules in webpack

Using `enhanced-resolve`, webpack can resolve three kinds of file paths:

Absolute paths

```
import "/home/me/file";

import "C:\\Users\\me\\file";
```

Since we already have the absolute path to the file, no further resolution is required.

Relative paths

```
import "../src/file1";
import "./file2";
```

In this case, the directory of the resource file where the `import` or `require` occurs is taken to be the context directory. The relative path specified in the `import/require` is joined to this context path to produce the absolute path to the module.

Module paths

```
import "module";
import "module/lib/file";
```

Modules are searched for inside all directories specified in `resolve.modules`. You can replace the original module path by an alternate path by creating an alias for it using `resolve.alias` configuration option.

Once the path is resolved based on the above rule, the resolver checks to see if the path points to a file or a directory. If the path points to a file:

- If the path has a file extension, then the file is bundled straightaway.
- Otherwise, the file extension is resolved using the `resolve.extensions` option, which tells the resolver which extensions (eg - `.js`, `.jsx`) are acceptable for resolution.

If the path points to a folder, then the following steps are taken to find the right file with the right extension:

- If the folder contains a `package.json` file, then fields specified in `resolve.mainFields` configuration option are looked up in order, and the first such field in `package.json` determines the file path.
- If there is no `package.json` or if the main fields do not return a valid path, file names specified in the

`resolve.mainFiles` configuration option are looked for in order, to see if a matching filename exists in the imported/required directory .
- The file extension is then resolved in a similar way using the `resolve.extensions` option.

webpack provides reasonable defaults for these options depending on your build target.

Resolving Loaders

This follows the same rules as those specified for file resolution. But the `resolveLoader` configuration option can be used to have separate resolution rules for loaders.

Caching

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 off, then the cache gets purged before every compilation.

See Resolve API to learn more on the configuration options mentioned above.

Any time one file depends on another, webpack treats this as a *dependency*. This allows webpack to take non-code assets, such as images or web fonts, and also provide them as *dependencies* for your application.

When webpack processes your application, it starts from a list of modules defined on the command line or in its config file. Starting from these *entry points*, webpack recursively builds a *dependency graph* that includes every module your application needs, then packages all of those modules into a small number of *bundles* - often, just one - to be loaded by the browser.

T> Bundling your application is especially powerful for *HTTP/1.1* clients, as it minimizes the number of times your app has to wait while the browser starts a new request. For *HTTP/2*, you can also use Code Splitting and bundling through webpack for the best optimization.

In a typical application or site built with webpack, there are three main types of code:

1. The source code you, and maybe your team, have written.
2. Any third-party library or "vendor" code your source is dependent on.
3. A webpack runtime and *manifest* that conducts the interaction of all modules.

This article will focus on the last of these three parts, the runtime and in particular the manifest.

Runtime

As mentioned above, we'll only briefly touch on this. The runtime, along with the manifest data, is basically all the code webpack needs to connect your modularized application while it's running in the browser. It contains the loading and resolving logic needed to connect your modules as they interact. This includes connecting modules that have already been loaded into the browser as well as logic to lazy-load the ones that haven't.

Manifest

So, once your application hits the browser in the form of an `index.html` file, some bundles, and a variety of other assets, what does it look like? That `/src` directory you meticulously laid out is now gone, so how does webpack manage the interaction between all of your modules? This is where the manifest data comes in...

As the compiler enters, resolves, and maps out your application, it keeps detailed notes on all your modules. This collection of data is called the "Manifest" and it's what the runtime will use to resolve and load modules once they've been bundled and shipped to the browser. No matter which module syntax you have chosen, those `import` or `require` statements have now become `__webpack_require__` methods that point to module identifiers. Using the data in the manifest, the runtime will be able to find out where to retrieve the modules behind the identifiers.

The Problem

So now you have a little bit of insight about how webpack works behind the scenes. "But, how does this affect me?", you might ask. The simple answer is that most of the time it doesn't. The runtime will do its thing, utilizing the manifest, and everything will appear to just magically work once your application hits the browser. However, if you decide to improve your projects performance by utilizing browser caching, this process will all of a sudden become an important thing to understand.

By using content hashes within your bundle file names, you can indicate to the browser when the contents of a file has changed thus invalidating the cache. Once you start doing this though, you'll immediately notice some funny behavior. Certain hashes change even when their contents apparently does not. This is caused by the injection of the runtime and manifest which changes every build.

See the manifest section of our *Managing Built Files* guide to learn how to extract the manifest, and read the guides below to learn more about the intricacies of long term caching.

Because JavaScript can be written for both server and browser, webpack offers multiple deployment *targets* that you can set in your webpack configuration.

W> The webpack `target` property is not to be confused with the `output.libraryTarget` property. For more information see our guide on the `output` property.

Usage

To set the `target` property, you simply set the target value in your webpack config:

webpack.config.js

```
module.exports = {
  target: 'node'
};
```

In the example above, using `node` webpack will compile for usage in a Node.js-like environment (uses Node.js `require` to load chunks and not touch any built in modules like `fs` or `path`).

Each *target* has a variety of deployment/environment specific additions, support to fit its needs. See what targets are available.

?>Further expansion for other popular target values

Multiple Targets

Although webpack does **not** support multiple strings being passed into the `target` property, you can create an isomorphic library by bundling two separate configurations:

webpack.config.js

```
var path = require('path');
var serverConfig = {
  target: 'node',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'lib.node.js'
  }
  //…
};

var clientConfig = {
  target: 'web', // <=== can be omitted as default is 'web'
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'lib.js'
  }
  //…
};

module.exports = [ serverConfig, clientConfig ];
```

The example above will create a `lib.js` and `lib.node.js` file in your `dist` folder.

Resources

As seen from the options above there are multiple different deployment *targets* that you can choose from. Below is a list of examples, and resources that you can refer to.

- **compare-webpack-target-bundles**: A great resource for testing and viewing different webpack *targets*. Also great for bug reporting.
- **Boilerplate of Electron-React Application**: A good example of a build process for electron's main process and renderer process.

?> Need to find up to date examples of these webpack targets being used in live code or boilerplates.

Hot Module Replacement (HMR) exchanges, adds, or removes modules while an application is running, without a full reload. This can significantly speed up development in a few ways:

- Retain application state which is lost during a full reload.
- Save valuable development time by only updating what's changed.
- Tweak styling faster -- almost comparable to changing styles in the browser's debugger.

How It Works

Let's go through some different viewpoints to understand exactly how HMR works...

In the Application

The following steps allow modules to be swapped in and out of an application:

1. The application asks the HMR runtime to check for updates.
2. The runtime asynchronously downloads the updates and notifies the application.
3. The application then asks the runtime to apply the updates.
4. The runtime synchronously applies the updates.

You can set up HMR so that this process happens automatically, or you can choose to require user interaction for updates to occur.

In the Compiler

In addition to normal assets, the compiler needs to emit an "update" to allow updating from previous version to the new version. The "update" consists of two parts:

1. The updated manifest (JSON)
2. One or more updated chunks (JavaScript)

The manifest contains the new compilation hash and a list of all updated chunks. Each of these chunks contains the new code for all updated modules (or a flag indicating that the module was removed).

The compiler ensures that module IDs and chunk IDs are consistent between these builds. It typically stores these IDs in memory (e.g. with webpack-dev-server), but it's also possible to store them in a JSON file.

In a Module

HMR is an opt-in feature that only affects modules containing HMR code. One example would be patching styling through the `style-loader`. In order for patching to work, the `style-loader` implements the HMR interface; when it receives an update through HMR, it replaces the old styles with the new ones.

Similarly, when implementing the HMR interface in a module, you can describe what should happen when the module is updated. However, in most cases, it's not mandatory to write HMR code in every module. If a module has no HMR handlers, the update bubbles up. This means that a single handler can update a complete module tree. If a single module from the tree is updated, the entire set of dependencies is reloaded.

See the HMR API page for details on the `module.hot` interface.

In the Runtime

Here things get a bit more technical... if you're not interested in the internals, feel free to jump to the HMR API page or HMR guide.

For the module system runtime, additional code is emitted to track module `parents` and `children`. On the management side, the runtime supports two methods: `check` and `apply`.

A `check` makes an HTTP request to the update manifest. If this request fails, there is no update available. If it succeeds, the list of updated chunks is compared to the list of currently loaded chunks. For each loaded chunk, the corresponding update chunk is downloaded. All module updates are stored in the runtime. When all update chunks have been downloaded and are ready to be applied, the runtime switches into the `ready` state.

The `apply` method flags all updated modules as invalid. For each invalid module, there needs to be an update handler in the module or in its parent(s). Otherwise, the invalid flag bubbles up and invalidates parent(s) as well. Each bubble continues until the app's entry point or a module with an update handler is reached (whichever comes first). If it bubbles up from an entry point, the process fails.

Afterwards, all invalid modules are disposed (via the dispose handler) and unloaded. The current hash is then updated and all `accept` handlers are called. The runtime switches back to the `idle` state and everything continues as normal.

Get Started

HMR can be used in development as a LiveReload replacement. webpack-dev-server supports a `hot` mode in which it tries to update with HMR before trying to reload the whole page. See the Hot Module Replacement guide for details.

T> As with many other features, webpack's power lies in its customizability. There are *many* ways of configuring HMR depending on the needs of a particular project. However, for most purposes, `webpack-dev-server` is a good fit and will allow you to get started with HMR quickly.

webpack is fed via a configuration object. It is passed in one of two ways depending on how you are using webpack: through the terminal or via Node.js. All the available configuration options are specified below.

T> New to webpack? Check out our guide to some of webpack's core concepts to get started!

T> Notice that throughout the configuration we use Node's built-in path module and prefix it with the __dirname global. This prevents file path issues between operating systems and allows relative paths to work as expected. See this section for more info on POSIX vs. Windows paths.

Options

Click on the name of each option in the configuration code below to jump to the detailed documentation. Also note that the items with arrows can be expanded to show more examples and, in some cases, more advanced configuration.

webpack.config.js

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

module.exports = {
  <details><summary>[mode](/concepts/mode): "production", // "production" | "development" | "none"</summary>
  [mode](/concepts/mode): "production", // enable many optimizations for production builds
  [mode](/concepts/mode): "development", // enabled useful tools for development
  [mode](/concepts/mode): "none", // no defaults
  </details>
  // Chosen mode tells webpack to use its built-in optimizations accordingly.

  <details><summary>[entry](/configuration/entry-context#entry): "./app/entry", // string | object | array</sum
mary>
  [entry](/configuration/entry-context#entry): ["./app/entry1", "./app/entry2"],
  [entry](/configuration/entry-context#entry): {
    a: "./app/entry-a",
    b: ["./app/entry-b1", "./app/entry-b2"]
  },
  </details>
  // Here the application starts executing
  // and webpack starts bundling

  [output](/configuration/output): {
    // options related to how webpack emits results

    [path](/configuration/output#output-path): path.resolve(__dirname, "dist"), // string
    // the target directory for all output files
    // must be an absolute path (use the Node.js path module)

    <details><summary>[filename](/configuration/output#output-filename): "bundle.js", // string</summary>
    [filename](/configuration/output#output-filename): "[name].js", // for multiple entry points
    [filename](/configuration/output#output-filename): "[chunkhash].js", // for [long term caching](/guides/cac
hing)
    </details>
    // the filename template for entry chunks

    <details><summary>[publicPath](/configuration/output#output-publicpath): "/assets/", // string</summary>
    [publicPath](/configuration/output#output-publicpath): "",
    [publicPath](/configuration/output#output-publicpath): "https://cdn.example.com/",
    </details>
    // the url to the output directory resolved relative to the HTML page

    [library](/configuration/output#output-library): "MyLibrary", // string,
    // the name of the exported library

    <details><summary>[libraryTarget](/configuration/output#output-librarytarget): "umd", // universal module d
efinition</summary>
        [libraryTarget](/configuration/output#output-librarytarget): "umd2", // universal module definition
        [libraryTarget](/configuration/output#output-librarytarget): "commonjs2", // exported with module.expor
```

```
ts
        [libraryTarget](/configuration/output#output-librarytarget): "commonjs", // exported as properties to e
xports
        [libraryTarget](/configuration/output#output-librarytarget): "amd", // defined with AMD defined method
        [libraryTarget](/configuration/output#output-librarytarget): "this", // property set on this
        [libraryTarget](/configuration/output#output-librarytarget): "var", // variable defined in root scope
        [libraryTarget](/configuration/output#output-librarytarget): "assign", // blind assignment
        [libraryTarget](/configuration/output#output-librarytarget): "window", // property set to window object
        [libraryTarget](/configuration/output#output-librarytarget): "global", // property set to global object
        [libraryTarget](/configuration/output#output-librarytarget): "jsonp", // jsonp wrapper
    </details>
    // the type of the exported library

    <details><summary>/* Advanced output configuration (click to show) */</summary>

    [pathinfo](/configuration/output#output-pathinfo): true, // boolean
    // include useful path info about modules, exports, requests, etc. into the generated code

    [chunkFilename](/configuration/output#output-chunkfilename): "[id].js",
    [chunkFilename](/configuration/output#output-chunkfilename): "[chunkhash].js", // for [long term caching](/
guides/caching)
    // the filename template for additional chunks

    [jsonpFunction](/configuration/output#output-jsonpfunction): "myWebpackJsonp", // string
    // name of the JSONP function used to load chunks

    [sourceMapFilename](/configuration/output#output-sourcemapfilename): "[file].map", // string
    [sourceMapFilename](/configuration/output#output-sourcemapfilename): "sourcemaps/[file].map", // string
    // the filename template of the source map location

    [devtoolModuleFilenameTemplate](/configuration/output#output-devtoolmodulefilenametemplate): "webpack:///[r
esource-path]", // string
    // the name template for modules in a devtool

    [devtoolFallbackModuleFilenameTemplate](/configuration/output#output-devtoolfallbackmodulefilenametemplate)
: "webpack:///[resource-path]?[hash]", // string
    // the name template for modules in a devtool (used for conflicts)

    [umdNamedDefine](/configuration/output#output-umdnameddefine): true, // boolean
    // use a named AMD module in UMD library

    [crossOriginLoading](/configuration/output#output-crossoriginloading): "use-credentials", // enum
    [crossOriginLoading](/configuration/output#output-crossoriginloading): "anonymous",
    [crossOriginLoading](/configuration/output#output-crossoriginloading): false,
    // specifies how cross origin request are issued by the runtime

    <details><summary>/* Expert output configuration (on own risk) */</summary>

    [devtoolLineToLine](/configuration/output#output-devtoollinetoline): {
      test: /\.jsx$/
    },
    // use a simple 1:1 mapped SourceMaps for these modules (faster)

    [hotUpdateMainFilename](/configuration/output#output-hotupdatemainfilename): "[hash].hot-update.json", // s
tring
    // filename template for HMR manifest

    [hotUpdateChunkFilename](/configuration/output#output-hotupdatechunkfilename): "[id].[hash].hot-update.js",
 // string
    // filename template for HMR chunks

    [sourcePrefix](/configuration/output#output-sourceprefix): "\t", // string
    // prefix module sources in bundle for better readablitity
    </details>
    </details>
  },

  [module](/configuration/module): {
    // configuration regarding modules
```

```
    [rules](/configuration/module#module-rules): [
      // rules for modules (configure loaders, parser options, etc.)

      {
        [test](/configuration/module#rule-test): /\.jsx?$/,
        [include](/configuration/module#rule-include): [
          path.resolve(__dirname, "app")
        ],
        [exclude](/configuration/module#rule-exclude): [
          path.resolve(__dirname, "app/demo-files")
        ],
        // these are matching conditions, each accepting a regular expression or string
        // test and include have the same behavior, both must be matched
        // exclude must not be matched (takes preferrence over test and include)
        // Best practices:
        // - Use RegExp only in test and for filename matching
        // - Use arrays of absolute paths in include and exclude
        // - Try to avoid exclude and prefer include

        [issuer](/configuration/module#rule-issuer): { test, include, exclude },
        // conditions for the issuer (the origin of the import)

        [enforce](/configuration/module#rule-enforce): "pre",
        [enforce](/configuration/module#rule-enforce): "post",
        // flags to apply these rules, even if they are overridden (advanced option)

        [loader](/configuration/module#rule-loader): "babel-loader",
        // the loader which should be applied, it'll be resolved relative to the context
        // -loader suffix is no longer optional in webpack2 for clarity reasons
        // see [webpack 1 upgrade guide](/guides/migrating)

        [options](/configuration/module#rule-options-rule-query): {
          presets: ["es2015"]
        },
        // options for the loader
      },

      {
        [test](/configuration/module#rule-test): /\.html$/,

        [use](/configuration/module#rule-use): [
          // apply multiple loaders and options
          "htmllint-loader",
          {
            loader: "html-loader",
            options: {
              /* ... */
            }
          }
        ]
      },

      { [oneOf](/configuration/module#rule-oneof): [ /* rules */ ] },
      // only use one of these nested rules

      { [rules](/configuration/module#rule-rules): [ /* rules */ ] },
      // use all of these nested rules (combine with conditions to be useful)

      { [resource](/configuration/module#rule-resource): { [and](/configuration/module#condition): [ /* conditi
ons */ ] } },
      // matches only if all conditions are matched

      { [resource](/configuration/module#rule-resource): { [or](/configuration/module#condition): [ /* conditio
ns */ ] } },
      { [resource](/configuration/module#rule-resource): [ /* conditions */ ] },
      // matches if any condition is matched (default for arrays)

      { [resource](/configuration/module#rule-resource): { [not](/configuration/module#condition): /* condition
```

```
*/ } }
     // matches if the condition is not matched
   ],

   <details><summary>/* Advanced module configuration (click to show) */</summary>

   [noParse](/configuration/module#module-noparse): [
     /special-library\.js$/
   ],
   // do not parse this module

   unknownContextRequest: ".",
   unknownContextRecursive: true,
   unknownContextRegExp: /^\.\/.*$/,
   unknownContextCritical: true,
   exprContextRequest: ".",
   exprContextRegExp: /^\.\/.*$/,
   exprContextRecursive: true,
   exprContextCritical: true,
   wrappedContextRegExp: /.*/,
   wrappedContextRecursive: true,
   wrappedContextCritical: false,
   // specifies default behavior for dynamic requests
   </details>
 },

 [resolve](/configuration/resolve): {
   // options for resolving module requests
   // (does not apply to resolving to loaders)

   [modules](/configuration/resolve#resolve-modules): [
     "node_modules",
     path.resolve(__dirname, "app")
   ],
   // directories where to look for modules

   [extensions](/configuration/resolve#resolve-extensions): [".js", ".json", ".jsx", ".css"],
   // extensions that are used

   [alias](/configuration/resolve#resolve-alias): {
     // a list of module name aliases

     "module": "new-module",
     // alias "module" -> "new-module" and "module/path/file" -> "new-module/path/file"

     "only-module$": "new-module",
     // alias "only-module" -> "new-module", but not "only-module/path/file" -> "new-module/path/file"

     "module": path.resolve(__dirname, "app/third/module.js"),
     // alias "module" -> "./app/third/module.js" and "module/file" results in error
     // modules aliases are imported relative to the current context
   },
   <details><summary>/* alternative alias syntax (click to show) */</summary>
   [alias](/configuration/resolve#resolve-alias): [
     {
       name: "module",
       // the old request

       alias: "new-module",
       // the new request

       onlyModule: true
       // if true only "module" is aliased
       // if false "module/inner/path" is also aliased
     }
   ],
   </details>

   <details><summary>/* Advanced resolve configuration (click to show) */</summary>
```

```
    [symlinks](/configuration/resolve#resolve-symlinks): true,
    // follow symlinks to new location

    [descriptionFiles](/configuration/resolve#resolve-descriptionfiles): ["package.json"],
    // files that are read for package description

    [mainFields](/configuration/resolve#resolve-mainfields): ["main"],
    // properties that are read from description file
    // when a folder is requested

    [aliasFields](/configuration/resolve#resolve-aliasfields): ["browser"],
    // properites that are read from description file
    // to alias requests in this package

    [enforceExtension](/configuration/resolve#resolve-enforceextension): false,
    // if true request must not include an extensions
    // if false request may already include an extension

    [moduleExtensions](/configuration/resolve#resolveloader-moduleextensions): ["-module"],
    [enforceModuleExtension](/configuration/resolve#resolve-enforcemoduleextension): false,
    // like extensions/enforceExtension but for module names instead of files

    [unsafeCache](/configuration/resolve#resolve-unsafecache): true,
    [unsafeCache](/configuration/resolve#resolve-unsafecache): {},
    // enables caching for resolved requests
    // this is unsafe as folder structure may change
    // but performance improvement is really big

    [cachePredicate](/configuration/resolve#resolve-cachepredicate): (path, request) => true,
    // predicate function which selects requests for caching

    [plugins](/configuration/resolve#resolve-plugins): [
      // ...
    ]
    // additional plugins applied to the resolver
    </details>
  },

  [performance](/configuration/performance): {
    <details><summary>[hints](/configuration/performance#performance-hints): "warning", // enum </summary>
    [hints](/configuration/performance#performance-hints): "error", // emit errors for perf hints
    [hints](/configuration/performance#performance-hints): false, // turn off perf hints
    </details>
    [maxAssetSize](/configuration/performance#performance-maxassetsize): 200000, // int (in bytes),
    [maxEntrypointSize](/configuration/performance#performance-maxentrypointsize): 400000, // int (in bytes)
    [assetFilter](/configuration/performance#performance-assetfilter): function(assetFilename) {
      // Function predicate that provides asset filenames
      return assetFilename.endsWith('.css') || assetFilename.endsWith('.js');
    }
  },

  <details><summary>[devtool](/configuration/devtool): "source-map", // enum </summary>
  [devtool](/configuration/devtool): "inline-source-map", // inlines SourceMap into original file
  [devtool](/configuration/devtool): "eval-source-map", // inlines SourceMap per module
  [devtool](/configuration/devtool): "hidden-source-map", // SourceMap without reference in original file
  [devtool](/configuration/devtool): "cheap-source-map", // cheap-variant of SourceMap without module mappings
  [devtool](/configuration/devtool): "cheap-module-source-map", // cheap-variant of SourceMap with module mappi
ngs
  [devtool](/configuration/devtool): "eval", // no SourceMap, but named modules. Fastest at the expense of deta
il.
  </details>
  // enhance debugging by adding meta info for the browser devtools
  // source-map most detailed at the expense of build speed.

  [context](/configuration/entry-context#context): __dirname, // string (absolute path!)
  // the home directory for webpack
  // the [entry](/configuration/entry-context) and [module.rules.loader](/configuration/module#rule-loader) opt
ion
```

```
//   is resolved relative to this directory

<details><summary>[target](/configuration/target): "web", // enum</summary>
[target](/configuration/target): "webworker", // WebWorker
[target](/configuration/target): "node", // Node.js via require
[target](/configuration/target): "async-node", // Node.js via fs and vm
[target](/configuration/target): "node-webkit", // nw.js
[target](/configuration/target): "electron-main", // electron, main process
[target](/configuration/target): "electron-renderer", // electron, renderer process
[target](/configuration/target): (compiler) => { /* ... */ }, // custom
</details>
// the environment in which the bundle should run
// changes chunk loading behavior and available modules

<details><summary>[externals](/configuration/externals): ["react", /^@angular\//],</summary>
[externals](/configuration/externals): "react", // string (exact match)
[externals](/configuration/externals): /^[a-z\-]+($|\/)/, // Regex
[externals](/configuration/externals): { // object
  angular: "this angular", // this["angular"]
  react: { // UMD
    commonjs: "react",
    commonjs2: "react",
    amd: "react",
    root: "React"
  }
},
[externals](/configuration/externals): (request) => { /* ... */ return "commonjs " + request }
</details>
// Don't follow/bundle these modules, but request them at runtime from the environment

<details><summary>[stats](/configuration/stats): "errors-only",</summary>
[stats](/configuration/stats): { //object
  assets: true,
  colors: true,
  errors: true,
  errorDetails: true,
  hash: true,
  // ...
},
</details>
// lets you precisely control what bundle information gets displayed

[devServer](/configuration/dev-server): {
  proxy: { // proxy URLs to backend development server
    '/api': 'http://localhost:3000'
  },
  contentBase: path.join(__dirname, 'public'), // boolean | string | array, static file location
  compress: true, // enable gzip compression
  historyApiFallback: true, // true for index.html upon 404, object for multiple paths
  hot: true, // hot module replacement. Depends on HotModuleReplacementPlugin
  https: false, // true for self-signed, object for cert authority
  noInfo: true, // only errors & warns on hot reload
  // ...
},

[plugins](plugins): [
  // ...
],
// list of additional plugins


<details><summary>/* Advanced configuration (click to show) */</summary>

[resolveLoader](/configuration/resolve#resolveloader): { /* same as resolve */ }
// separate resolve options for loaders

[parallelism](other-options#parallelism): 1, // number
// limit the number of parallel processed modules
```

```
  [profile](other-options#profile): true, // boolean
  // capture timing information

  [bail](other-options#bail): true, //boolean
  // fail out on the first error instead of tolerating it.

  [cache](other-options#cache): false, // boolean
  // disable/enable caching

  [watch](watch#watch): true, // boolean
  // enables watching

  [watchOptions](watch#watchoptions): {
    [aggregateTimeout](watch#watchoptions-aggregatetimeout): 1000, // in ms
    // aggregates multiple changes to a single rebuild

    [poll](watch#watchoptions-poll): true,
    [poll](watch#watchoptions-poll): 500, // intervall in ms
    // enables polling mode for watching
    // must be used on filesystems that doesn't notify on change
    // i. e. nfs shares
  },

  [node](node): {
    // Polyfills and mocks to run Node.js-
    // environment code in non-Node environments.

    [console](node#node-console): false, // boolean | "mock"
    [global](node#node-global): true, // boolean | "mock"
    [process](node#node-process): true, // boolean
    [__filename](node#node-__filename): "mock", // boolean | "mock"
    [__dirname](node#node-__dirname): "mock", // boolean | "mock"
    [Buffer](node#node-buffer): true, // boolean | "mock"
    [setImmediate](node#node-setimmediate): true // boolean | "mock" | "empty"
  },

  [recordsPath](other-options#recordspath): path.resolve(__dirname, "build/records.json"),
  [recordsInputPath](other-options#recordsinputpath): path.resolve(__dirname, "build/records.json"),
  [recordsOutputPath](other-options#recordsoutputpath): path.resolve(__dirname, "build/records.json"),
  // TODO

  </details>
}
```

webpack accepts configuration files written in multiple programming and data languages. The list of supported file extensions can be found at the node-interpret package. Using node-interpret, webpack can handle many different types of configuration files.

TypeScript

To write the webpack configuration in TypeScript, you would first install the necessary dependencies:

```
npm install --save-dev typescript ts-node @types/node @types/webpack
```

and then proceed to write your configuration:

webpack.config.ts

```
import path from 'path';
import webpack from 'webpack';

const config: webpack.Configuration = {
  mode: 'production',
  entry: './foo.js',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'foo.bundle.js'
  }
};

export default config;
```

Above sample assumes version >= 2.7 or newer of TypeScript is used with the new `esModuleInterop` and `allowSyntheticDefaultImports` compiler options in your `tsconfig.json` file.

Note that you'll also need to check your `tsconfig.json` file. If the module in `compilerOptions` in `tsconfig.json` is `commonjs`, the setting is complete, else webpack will fail with an error. This occurs because `ts-node` does not support any module syntax other than `commonjs`.

There are two solutions to this issue:

- Modify `tsconfig.json`.
- Install `tsconfig-paths`.

The **first option** is to open your `tsconfig.json` file and look for `compilerOptions`. Set `target` to `"ES5"` and `module` to `"CommonJS"` (or completely remove the `module` option).

The **second option** is to install the `tsconfig-paths` package:

```
npm install --save-dev tsconfig-paths
```

And create a separate TypeScript configuration specifically for your webpack configs:

tsconfig-for-webpack-config.json

```
{
  "compilerOptions": {
    "module": "commonjs",
    "target": "es5"
  }
}
```

T> `ts-node` can resolve a `tsconfig.json` file using the environment variable provided by `tsconfig-path`.

Then set the environment variable `process.env.TS_NODE_PROJECT` provided by `tsconfig-path` like so:

package.json

```
{
  "scripts": {
    "build": "TS_NODE_PROJECT=\"tsconfig-for-webpack-config.json\" webpack"
  }
}
```

CoffeeScript

Similarly, to use CoffeeScript, you would first install the necessary dependencies:

```
npm install --save-dev coffee-script
```

and then proceed to write your configuration:

webpack.config.coffee

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

config =
  mode: 'production'
  entry: './path/to/my/entry/file.js'
  output:
    path: path.resolve(__dirname, 'dist')
    filename: 'my-first-webpack.bundle.js'
  module: rules: [ {
    test: /\.(js|jsx)$/
    use: 'babel-loader'
  } ]
  plugins: [
    new (webpack.optimize.UglifyJsPlugin)
    new HtmlWebpackPlugin(template: './src/index.html')
  ]

module.exports = config
```

Babel and JSX

In the example below JSX (React JavaScript Markup) and Babel are used to create a JSON Configuration that webpack can understand.

> Courtesy of Jason Miller

First install the necessary dependencies:

```
npm install --save-dev babel-register jsxobj babel-preset-es2015
```

.babelrc

```
{
```

```
  "presets": [ "es2015" ]
}
```

webpack.config.babel.js

```
import jsxobj from 'jsxobj';

// example of an imported plugin
const CustomPlugin = config => ({
  ...config,
  name: 'custom-plugin'
});

export default (
  <webpack target="web" watch mode="production">
    <entry path="src/index.js" />
    <resolve>
      <alias {...{
        react: 'preact-compat',
        'react-dom': 'preact-compat'
      }} />
    </resolve>
    <plugins>
      <uglify-js opts={{
        compression: true,
        mangle: false
      }} />
      <CustomPlugin foo="bar" />
    </plugins>
  </webpack>
);
```

W> If you are using Babel elsewhere and have `modules` set to `false`, you will have to either maintain two separate `.babelrc` files or use `const jsxobj = require('jsxobj');` and `module.exports` instead of the new `import` and `export` syntax. This is because while Node does support many new ES6 features, they don't yet support ES6 module syntax.

Besides exporting a single config object, there are a few more ways that cover other needs as well.

Exporting a Function

Eventually you will find the need to disambiguate in your `webpack.config.js` between development and production builds. You have (at least) two options:

One option is to export a function from your webpack config instead of exporting an object. The function will be invoked with two arguments:

- An environment as the first parameter. See the environment options CLI documentation for syntax examples.
- An options map (`argv`) as the second parameter. This describes the options passed to webpack, with keys such as `output-filename` and `optimize-minimize` .

```
-module.exports = {
+module.exports = function(env, argv) {
+  return {
+    mode: env.production ? 'production' : 'development',
+    devtool: env.production ? 'source-maps' : 'eval',
     plugins: [
       new webpack.optimize.UglifyJsPlugin({
+        compress: argv['optimize-minimize'] // only if -p or --optimize-minimize were passed
       })
     ]
+  };
};
```

Exporting a Promise

webpack will run the function exported by the configuration file and wait for a Promise to be returned. Handy when you need to asynchronously load configuration variables.

```
module.exports = () => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve({
        entry: './app.js',
        /* ... */
      })
    }, 5000)
  })
}
```

Exporting multiple configurations

Instead of exporting a single configuration object/function, you may export multiple configurations (multiple functions are supported since webpack 3.1.0). When running webpack, all configurations are built. For instance, this is useful for bundling a library for multiple targets such as AMD and CommonJS:

```
module.exports = [{
  output: {
    filename: './dist-amd.js',
    libraryTarget: 'amd'
  },
  entry: './app.js',
  mode: 'production',
```

```
}, {
  output: {
    filename: './dist-commonjs.js',
    libraryTarget: 'commonjs'
  },
  entry: './app.js',
  mode: 'production',
}]
```

The entry object is where webpack looks to start building the bundle. The context is an absolute string to the directory that contains the entry files.

`context`

`string`

The base directory, an **absolute path**, for resolving entry points and loaders from configuration.

```
context: path.resolve(__dirname, "app")
```

By default the current directory is used, but it's recommended to pass a value in your configuration. This makes your configuration independent from CWD (current working directory).

`entry`

`string | [string] | object { <key>: string | [string] } | (function: () => string | [string] | object { <key>: string | [string] })`

The point or points to enter the application. At this point the application starts executing. If an array is passed all items will be executed.

A dynamically loaded module is **not** an entry point.

Simple rule: one entry point per HTML page. SPA: one entry point, MPA: multiple entry points.

```
entry: {
  home: "./home.js",
  about: "./about.js",
  contact: "./contact.js"
}
```

Naming

If a string or array of strings is passed, the chunk is named `main`. If an object is passed, each key is the name of a chunk, and the value describes the entrypoint for the chunk.

Dynamic entry

```
entry: () => './demo'
```

or

```
entry: () => new Promise((resolve) => resolve(['./demo', './demo2']))
```

When combining with the `output.library` option: If an array is passed only the last item is exported.

The top-level `output` key contains set of options instructing webpack on how and where it should output your bundles, assets and anything else you bundle or load with webpack.

`output.auxiliaryComment`

`string` `object`

When used in tandem with `output.library` and `output.libraryTarget`, this option allows users to insert comments within the export wrapper. To insert the same comment for each `libraryTarget` type, set `auxiliaryComment` to a string:

```
output: {
  library: "someLibName",
  libraryTarget: "umd",
  filename: "someLibName.js",
  auxiliaryComment: "Test Comment"
}
```

which will yield the following:

```
(function webpackUniversalModuleDefinition(root, factory) {
  // Test Comment
  if(typeof exports === 'object' && typeof module === 'object')
    module.exports = factory(require("lodash"));
  // Test Comment
  else if(typeof define === 'function' && define.amd)
    define(["lodash"], factory);
  // Test Comment
  else if(typeof exports === 'object')
    exports["someLibName"] = factory(require("lodash"));
  // Test Comment
  else
    root["someLibName"] = factory(root["_"]);
})(this, function(__WEBPACK_EXTERNAL_MODULE_1__) {
  // ...
});
```

For fine-grained control over each `libraryTarget` comment, pass an object:

```
auxiliaryComment: {
  root: "Root Comment",
  commonjs: "CommonJS Comment",
  commonjs2: "CommonJS2 Comment",
  amd: "AMD Comment"
}
```

`output.chunkFilename`

`string` `function`

This option determines the name of non-entry chunk files. See `output.filename` option for details on the possible values.

Note that these filenames need to be generated at runtime to send the requests for chunks. Because of this, placeholders like `[name]` and `[chunkhash]` need to add a mapping from chunk id to placeholder value to the output bundle with the webpack runtime. This increases the size and may invalidate the bundle when placeholder value for any chunk changes.

By default `[id].js` is used or a value inferred from `output.filename` (`[name]` is replaced with `[id]` or `[id].` is prepended).

`output.chunkLoadTimeout`

`integer`

Number of milliseconds before chunk request expires, defaults to 120 000. This option is supported since webpack 2.6.0.

`output.crossOriginLoading`

`boolean` `string`

Only used when `target` is web, which uses JSONP for loading on-demand chunks, by adding script tags.

Enable cross-origin loading of chunks. The following values are accepted...

`crossOriginLoading: false` - Disable cross-origin loading (default)

`crossOriginLoading: "anonymous"` - Enable cross-origin loading **without credentials**

`crossOriginLoading: "use-credentials"` - Enable cross-origin loading **with credentials**

`output.jsonpScriptType`

`string`

Allows customization of the `script` type webpack injects `script` tags into the DOM to download async chunks. The following options are available:

- `"text/javascript"` (default)
- `"module"` : Use with ES6 ready code.

`output.devtoolFallbackModuleFilenameTemplate`

`string | function(info)`

A fallback used when the template string or function above yields duplicates.

See `output.devtoolModuleFilenameTemplate` .

`output.devtoolLineToLine`

`boolean | object`

> Avoid using this option as it is **deprecated** and will soon be removed.

Enables line to line mapping for all or some modules. This produces a simple source map where each line of the generated source is mapped to the same line of the original source. This is a performance optimization and should only be used if all input lines match generated lines.

Pass a boolean to enable or disable this feature for all modules (defaults to `false`). An object with `test` , `include` , `exclude` is also allowed. For example, to enable this feature for all javascript files within a certain directory:

```
devtoolLineToLine: { test: /\.js$/, include: 'src/utilities' }
```

`output.devtoolModuleFilenameTemplate`

`string | function(info)`

This option is only used when `devtool` uses an options which requires module names.

Customize the names used in each source map's `sources` array. This can be done by passing a template string or function. For example, when using `devtool: 'eval'`, this is the default:

```
devtoolModuleFilenameTemplate: "webpack://[namespace]/[resource-path]?[loaders]"
```

The following substitutions are available in template strings (via webpack's internal `ModuleFilenameHelpers`):

Template	Description
[absolute-resource-path]	The absolute filename
[all-loaders]	Automatic and explicit loaders and params up to the name of the first loader
[hash]	The hash of the module identifier
[id]	The module identifier
[loaders]	Explicit loaders and params up to the name of the first loader
[resource]	The path used to resolve the file and any query params used on the first loader
[resource-path]	The path used to resolve the file without any query params
[namespace]	The modules namespace. This is usually the library name when building as a library, empty otherwise

When using a function, the same options are available camel-cased via the `info` parameter:

```
devtoolModuleFilenameTemplate: info => {
  return `webpack:///${info.resourcePath}?${info.loaders}`
}
```

If multiple modules would result in the same name, `output.devtoolFallbackModuleFilenameTemplate` is used instead for these modules.

`output.devtoolNamespace`

`string`

This option determines the modules namespace used with the `output.devtoolModuleFilenameTemplate`. When not specified, it will default to the value of: `output.library`. It's used to prevent source file path collisions in sourcemaps when loading multiple libraries built with webpack.

For example, if you have 2 libraries, with namespaces `library1` and `library2`, which both have a file `./src/index.js` (with potentially different contents), they will expose these files as `webpack://library1/./src/index.js` and `webpack://library2/./src/index.js`.

`output.filename`

`string` `function`

This option determines the name of each output bundle. The bundle is written to the directory specified by the `output.path` option.

For a single `entry` point, this can be a static name.

```
filename: "bundle.js"
```

However, when creating multiple bundles via more than one entry point, code splitting, or various plugins, you should use one of the following substitutions to give each bundle a unique name...

Using entry name:

```
filename: "[name].bundle.js"
```

Using internal chunk id:

```
filename: "[id].bundle.js"
```

Using the unique hash generated for every build:

```
filename: "[name].[hash].bundle.js"
```

Using hashes based on each chunks' content:

```
filename: "[chunkhash].bundle.js"
```

Make sure to read the Caching guide for details. There are more steps involved than just setting this option.

Note this option is called filename but you are still allowed to use something like `"js/[name]/bundle.js"` to create a folder structure.

Note this option does not affect output files for on-demand-loaded chunks. For these files the `output.chunkFilename` option is used. Files created by loaders also aren't affected. In this case you would have to try the specific loader's available options.

The following substitutions are available in template strings (via webpack's internal `TemplatedPathPlugin`):

Template	Description
[hash]	The hash of the module identifier
[chunkhash]	The hash of the chunk content
[name]	The module name
[id]	The module identifier
[query]	The module query, i.e., the string following `?` in the filename

The lengths of `[hash]` and `[chunkhash]` can be specified using `[hash:16]` (defaults to 20). Alternatively, specify `output.hashDigestLength` to configure the length globally.

If using a function for this option, the function will be passed an object containing the substitutions in the table above.

T> When using the `ExtractTextWebpackPlugin`, use `[contenthash]` to obtain a hash of the extracted file (neither `[hash]` nor `[chunkhash]` work).

`output.hashDigest`

The encoding to use when generating the hash, defaults to `'hex'`. All encodings from Node.JS' `hash.digest` are supported.

`output.hashDigestLength`

The prefix length of the hash digest to use, defaults to `20`.

`output.hashFunction`

`string|function`

The hashing algorithm to use, defaults to `'md5'`. All functions from Node.JS' `crypto.createHash` are supported. Since `4.0.0-alpha2`, the `hashFunction` can now be a constructor to a custom hash function. You can provide a non-crypto hash function for performance reasons.

```
hashFunction: require('metrohash').MetroHash64
```

Make sure that the hashing function will have `update` and `digest` methods available.

`output.hashSalt`

An optional salt to update the hash via Node.JS' `hash.update`.

`output.hotUpdateChunkFilename`

`string` `function`

Customize the filenames of hot update chunks. See `output.filename` option for details on the possible values.

The only placeholders allowed here are `[id]` and `[hash]`, the default being:

```
hotUpdateChunkFilename: "[id].[hash].hot-update.js"
```

Here is no need to change it.

`output.hotUpdateFunction`

`function`

Only used when `target` is web, which uses JSONP for loading hot updates.

A JSONP function used to asynchronously load hot-update chunks.

For details see `output.jsonpFunction`.

`output.hotUpdateMainFilename`

`string` `function`

Customize the main hot update filename. See `output.filename` option for details on the possible values.

`[hash]` is the only available placeholder, the default being:

```
hotUpdateMainFilename: "[hash].hot-update.json"
```

Here is no need to change it.

`output.jsonpFunction`

`string`

Only used when `target` is web, which uses JSONP for loading on-demand chunks.

A JSONP function name used to asynchronously load chunks or join multiple initial chunks (CommonsChunkPlugin, AggressiveSplittingPlugin).

This needs to be changed if multiple webpack runtimes (from different compilation) are used on the same webpage.

If using the `output.library` option, the library name is automatically appended.

`output.library`

`string`

`string` or `object` (since webpack 3.1.0; for `libraryTarget: "umd"`)

How the value of the `output.library` is used depends on the value of the `output.libraryTarget` option; please refer to that section for the complete details. Note that the default option for `output.libraryTarget` is `var` , so if the following configuration option is used:

```
output: {
  library: "MyLibrary"
}
```

The variable `MyLibrary` will be bound with the return value of your entry file, if the resulting output is included as a script tag in an HTML page.

W> Note that if an `array` is provided as an `entry` point, only the last module in the array will be exposed. If an `object` is provided, it can exposed using an `array` syntax (see this example for details).

T> Read the authoring libraries guide guide for more information on `output.library` as well as `output.libraryTarget` .

`output.libraryExport`

`string` or `string[]` (since webpack 3.0.0)

> Default: `_entry_return_`

Configure which module or modules will be exposed via the `libraryTarget` . The default `_entry_return_` value is the namespace or default module returned by your entry file. The examples below demonstrate the effect of this config when using `libraryTarget: "var"` , but any target may be used.

The following configurations are supported:

`libraryExport: "default"` - The **default export of your entry point** will be assigned to the library target:

```
// if your entry has a default export of `MyDefaultModule`
```

```
var MyDefaultModule = _entry_return_.default;
```

`libraryExport: "MyModule"` - The **specified module** will be assigned to the library target:

```
var MyModule = _entry_return_.MyModule;
```

`libraryExport: ["MyModule", "MySubModule"]` - The array is interpreted as a **path to a module** to be assigned to the library target:

```
var MySubModule = _entry_return_.MyModule.MySubModule;
```

With the `libraryExport` configurations specified above, the resulting libraries could be utilized as such:

```
MyDefaultModule.doSomething();
MyModule.doSomething();
MySubModule.doSomething();
```

`output.libraryTarget`

`string`

> Default: `"var"`

Configure how the library will be exposed. Any one of the following options can be used. Please note that this option works in conjunction with the value assigned to `output.library`. For the following examples, it is assumed that this value is configured as `MyLibrary`.

T> Note that `_entry_return_` in the example code below is the value returned by the entry point. In the bundle itself, it is the output of the function that is generated by webpack from the entry point.

Expose a Variable

These options assign the return value of the entry point (e.g. whatever the entry point exported) to the name provided by `output.library` at whatever scope the bundle was included at.

`libraryTarget: "var"` - (default) When your library is loaded, the **return value of your entry point** will be assigned to a variable:

```
var MyLibrary = _entry_return_;

// In a separate script...
MyLibrary.doSomething();
```

W> When using this option, an empty `output.library` will result in no assignment.

`libraryTarget: "assign"` - This will generate an implied global which has the potential to reassign an existing value (use with caution).

```
MyLibrary = _entry_return_;
```

Be aware that if `MyLibrary` isn't defined earlier your library will be set in global scope.

W> When using this option, an empty `output.library` will result in a broken output bundle.

Expose Via Object Assignment

These options assign the return value of the entry point (e.g. whatever the entry point exported) to a specific object under the name defined by `output.library`.

If `output.library` is not assigned a non-empty string, the default behavior is that all properties returned by the entry point will be assigned to the object as defined for the particular `output.libraryTarget`, via the following code fragment:

```
(function(e, a) { for(var i in a) e[i] = a[i]; }(${output.libraryTarget}, _entry_return_)
```

W> Note that not setting a `output.library` will cause all properties returned by the entry point to be assigned to the given object; there are no checks against existing property names.

`libraryTarget: "this"` - The **return value of your entry point** will be assigned to this under the property named by `output.library`. The meaning of `this` is up to you:

```
this["MyLibrary"] = _entry_return_;

// In a separate script...
this.MyLibrary.doSomething();
MyLibrary.doSomething(); // if this is window
```

`libraryTarget: "window"` - The **return value of your entry point** will be assigned to the `window` object using the `output.library` value.

```
window["MyLibrary"] = _entry_return_;

window.MyLibrary.doSomething();
```

`libraryTarget: "global"` - The **return value of your entry point** will be assigned to the `global` object using the `output.library` value.

```
global["MyLibrary"] = _entry_return_;

global.MyLibrary.doSomething();
```

`libraryTarget: "commonjs"` - The **return value of your entry point** will be assigned to the `exports` object using the `output.library` value. As the name implies, this is used in CommonJS environments.

```
exports["MyLibrary"] = _entry_return_;

require("MyLibrary").doSomething();
```

Module Definition Systems

These options will result in a bundle that comes with a more complete header to ensure compatibility with various module systems. The `output.library` option will take on a different meaning under the following `output.libraryTarget` options.

`libraryTarget: "commonjs2"` - The **return value of your entry point** will be assigned to the `module.exports`. As the name implies, this is used in CommonJS environments:

```
module.exports = _entry_return_;
```

```
require("MyLibrary").doSomething();
```

Note that `output.library` is omitted, thus it is not required for this particular `output.libraryTarget`.

T> Wondering the difference between CommonJS and CommonJS2 is? While they are similar, there are some subtle differences between them that are not usually relevant in the context of webpack. (For further details, please read this issue.)

`libraryTarget: "amd"` - This will expose your library as an AMD module.

AMD modules require that the entry chunk (e.g. the first script loaded by the `<script>` tag) be defined with specific properties, such as `define` and `require` which is typically provided by RequireJS or any compatible loaders (such as almond). Otherwise, loading the resulting AMD bundle directly will result in an error like `define is not defined`.

So, with the following configuration...

```
output: {
  library: "MyLibrary",
  libraryTarget: "amd"
}
```

The generated output will be defined with the name "MyLibrary", i.e.

```
define("MyLibrary", [], function() {
  return _entry_return_;
});
```

The bundle can be included as part of a script tag, and the bundle can be invoked like so:

```
require(['MyLibrary'], function(MyLibrary) {
  // Do something with the library...
});
```

If `output.library` is undefined, the following is generated instead.

```
define([], function() {
  return _entry_return_;
});
```

This bundle will not work as expected, or not work at all (in the case of the almond loader) if loaded directly with a `<script>` tag. It will only work through a RequireJS compatible asynchronous module loader through the actual path to that file, so in this case, the `output.path` and `output.filename` may become important for this particular setup if these are exposed directly on the server.

`libraryTarget: "umd"` - This exposes your library under all the module definitions, allowing it to work with CommonJS, AMD and as global variable. Take a look at the UMD Repository to learn more.

In this case, you need the `library` property to name your module:

```
output: {
  library: "MyLibrary",
  libraryTarget: "umd"
}
```

And finally the output is:

```
(function webpackUniversalModuleDefinition(root, factory) {
```

```
  if(typeof exports === 'object' && typeof module === 'object')
    module.exports = factory();
  else if(typeof define === 'function' && define.amd)
    define([], factory);
  else if(typeof exports === 'object')
    exports["MyLibrary"] = factory();
  else
    root["MyLibrary"] = factory();
})(typeof self !== 'undefined' ? self : this, function() {
  return _entry_return_;
});
```

Note that omitting `library` will result in the assignment of all properties returned by the entry point be assigned directly to the root object, as documented under the object assignment section. Example:

```
output: {
  libraryTarget: "umd"
}
```

The output will be:

```
(function webpackUniversalModuleDefinition(root, factory) {
  if(typeof exports === 'object' && typeof module === 'object')
    module.exports = factory();
  else if(typeof define === 'function' && define.amd)
    define([], factory);
  else {
    var a = factory();
    for(var i in a) (typeof exports === 'object' ? exports : root)[i] = a[i];
  }
})(typeof self !== 'undefined' ? self : this, function() {
  return _entry_return_;
});
```

Since webpack 3.1.0, you may specify an object for `library` for differing names per targets:

```
output: {
  library: {
    root: "MyLibrary",
    amd: "my-library",
    commonjs: "my-common-library"
  },
  libraryTarget: "umd"
}
```

Module proof library.

Other Targets

`libraryTarget: "jsonp"` - This will wrap the return value of your entry point into a jsonp wrapper.

```
MyLibrary(_entry_return_);
```

The dependencies for your library will be defined by the `externals` config.

`output.path`

`string`

The output directory as an **absolute** path.

```
path: path.resolve(__dirname, 'dist/assets')
```

Note that `[hash]` in this parameter will be replaced with an hash of the compilation. See the Caching guide for details.

`output.pathinfo`

`boolean`

Tell webpack to include comments in bundles with information about the contained modules. This option defaults to `false` and **should not** be used in production, but it's very useful in development when reading the generated code.

```
pathinfo: true
```

Note it also adds some info about tree shaking to the generated bundle.

`output.publicPath`

`string` `function`

This is an important option when using on-demand-loading or loading external resources like images, files, etc. If an incorrect value is specified you'll receive 404 errors while loading these resources.

This option specifies the **public URL** of the output directory when referenced in a browser. A relative URL is resolved relative to the HTML page (or `<base>` tag). Server-relative URLs, protocol-relative URLs or absolute URLs are also possible and sometimes required, i. e. when hosting assets on a CDN.

The value of the option is prefixed to every URL created by the runtime or loaders. Because of this **the value of this option ends with** `/` in most cases.

The default value is an empty string `""`.

Simple rule: The URL of your `output.path` from the view of the HTML page.

```
path: path.resolve(__dirname, "public/assets"),
publicPath: "https://cdn.example.com/assets/"
```

For this configuration:

```
publicPath: "/assets/",
chunkFilename: "[id].chunk.js"
```

A request to a chunk will look like `/assets/4.chunk.js`.

A loader outputting HTML might emit something like this:

```
<link href="/assets/spinner.gif" />
```

or when loading an image in CSS:

```
background-image: url(/assets/spinner.gif);
```

The webpack-dev-server also takes a hint from `publicPath`, using it to determine where to serve the output files from.

Note that `[hash]` in this parameter will be replaced with an hash of the compilation. See the Caching guide for details.

Examples:

```
publicPath: "https://cdn.example.com/assets/", // CDN (always HTTPS)
publicPath: "//cdn.example.com/assets/", // CDN (same protocol)
publicPath: "/assets/", // server-relative
publicPath: "assets/", // relative to HTML page
publicPath: "../assets/", // relative to HTML page
publicPath: "", // relative to HTML page (same directory)
```

In cases where the `publicPath` of output files can't be known at compile time, it can be left blank and set dynamically at runtime in the entry file using the free variable `__webpack_public_path__`.

```
 __webpack_public_path__ = myRuntimePublicPath

// rest of your application entry
```

See this discussion for more information on `__webpack_public_path__`.

`output.sourceMapFilename`

`string`

This option is only used when `devtool` uses a SourceMap option which writes an output file.

Configure how source maps are named. By default `"[file].map"` is used.

The `[name]`, `[id]`, `[hash]` and `[chunkhash]` substitutions from #output-filename can be used. In addition to those, you can use substitutions listed below. The `[file]` placeholder is replaced with the filename of the original file. We recommend **only using the `[file]` placeholder**, as the other placeholders won't work when generating SourceMaps for non-chunk files.

Template	Description
[file]	The module filename
[filebase]	The module basename

`output.sourcePrefix`

`string`

Change the prefix for each line in the output bundles.

```
sourcePrefix: "\t"
```

Note by default an empty string is used. Using some kind of indentation makes bundles look more pretty, but will cause issues with multi-line strings.

There is no need to change it.

`output.strictModuleExceptionHandling`

`boolean`

Tell webpack to remove a module from the module instance cache (`require.cache`) if it throws an exception when it is `require` d.

It defaults to `false` for performance reasons.

When set to `false` , the module is not removed from cache, which results in the exception getting thrown only on the first `require` call (making it incompatible with node.js).

For instance, consider `module.js` :

```
throw new Error("error");
```

With `strictModuleExceptionHandling` set to `false` , only the first `require` throws an exception:

```
// with strictModuleExceptionHandling = false
require("module") // <- throws
require("module") // <- doesn't throw
```

Instead, with `strictModuleExceptionHandling` set to `true` , all `require` s of this module throw an exception:

```
// with strictModuleExceptionHandling = true
require("module") // <- throws
require("module") // <- also throw
```

`output.umdNamedDefine`

`boolean`

When using `libraryTarget: "umd"` , setting:

```
umdNamedDefine: true
```

will name the AMD module of the UMD build. Otherwise an anonymous `define` is used.

These options determine how the different types of modules within a project will be treated.

`module.noParse`

`RegExp | [RegExp]`

`RegExp | [RegExp] | function` (since webpack 3.0.0)

Prevent webpack from parsing any files matching the given regular expression(s). Ignored files **should not** have calls to `import`, `require`, `define` or any other importing mechanism. This can boost build performance when ignoring large libraries.

```
noParse: /jquery|lodash/

// since webpack 3.0.0
noParse: function(content) {
  return /jquery|lodash/.test(content);
}
```

`module.rules`

`array`

An array of Rules which are matched to requests when modules are created. These rules can modify how the module is created. They can apply loaders to the module, or modify the parser.

Rule

A Rule can be separated into three parts — Conditions, Results and nested Rules.

Rule Conditions

There are two input values for the conditions:

1. The resource: An absolute path to the file requested. It's already resolved according to the `resolve` rules.
2. The issuer: An absolute path to the file of the module which requested the resource. It's the location of the import.

Example: When we `import './style.css'` within `app.js`, the resource is `/path/to/style.css` and the issuer is `/path/to/app.js`.

In a Rule the properties `test`, `include`, `exclude` and `resource` are matched with the resource and the property `issuer` is matched with the issuer.

When using multiple conditions, all conditions must match.

W> Be careful! The resource is the *resolved* path of the file, which means symlinked resources are the real path *not* the symlink location. This is good to remember when using tools that symlink packages (like `npm link`), common conditions like `/node_modules/` may inadvertently miss symlinked files. Note that you can turn off symlink resolving (so that resources are resolved to the symlink path) via `resolve.symlinks`.

Rule results

Rule results are used only when the Rule condition matches.

There are two output values of a Rule:

1. Applied loaders: An array of loaders applied to the resource.
2. Parser options: An options object which should be used to create the parser for this module.

These properties affect the loaders: `loader`, `options`, `use`.

For compatibility also these properties: `query`, `loaders`.

The `enforce` property affects the loader category. Whether it's a normal, pre- or post- loader.

The `parser` property affects the parser options.

Nested rules

Nested rules can be specified under the properties `rules` and `oneOf`.

These rules are evaluated when the Rule condition matches.

`Rule.enforce`

Possible values: `"pre" | "post"`

Specifies the category of the loader. No value means normal loader.

There is also an additional category "inlined loader" which are loaders applied inline of the import/require.

All loaders are sorted in the order `pre, inline, normal, post` and used in this order.

All normal loaders can be omitted (overridden) by prefixing `!` in the request.

All normal and pre loaders can be omitted (overridden) by prefixing `-!` in the request.

All normal, post and pre loaders can be omitted (overridden) by prefixing `!!` in the request.

Inline loaders and `!` prefixes should not be used as they are non-standard. They may be use by loader generated code.

`Rule.exclude`

`Rule.exclude` is a shortcut to `Rule.resource.exclude`. If you supply a `Rule.exclude` option, you cannot also supply a `Rule.resource`. See `Rule.resource` and `Condition.exclude` for details.

`Rule.include`

`Rule.include` is a shortcut to `Rule.resource.include`. If you supply a `Rule.include` option, you cannot also supply a `Rule.resource`. See `Rule.resource` and `Condition.include` for details.

`Rule.issuer`

A `Condition` to match against the module that issued the request. In the following example, the `issuer` for the `a.js` request would be the path to the `index.js` file.

index.js

```
import A from './a.js'
```

This option can be used to apply loaders to the dependencies of a specific module or set of modules.

`Rule.loader`

`Rule.loader` is a shortcut to `Rule.use: [ { loader } ]`. See `Rule.use` and `UseEntry.loader` for details.

`Rule.loaders`

W> This option is **deprecated** in favor of `Rule.use`.

`Rule.loaders` is an alias to `Rule.use`. See `Rule.use` for details.

`Rule.oneOf`

An array of `Rules` from which only the first matching Rule is used when the Rule matches.

```
{
  test: /.css$/,
  oneOf: [
    {
      resourceQuery: /inline/, // foo.css?inline
      use: 'url-loader'
    },
    {
      resourceQuery: /external/, // foo.css?external
      use: 'file-loader'
    }
  ]
}
```

`Rule.options` / `Rule.query`

`Rule.options` and `Rule.query` are shortcuts to `Rule.use: [ { options } ]`. See `Rule.use` and `UseEntry.options` for details.

W> `Rule.query` is deprecated in favor of `Rule.options` and `UseEntry.options`.

`Rule.parser`

An object with parser options. All applied parser options are merged.

Parsers may inspect these options and disable or reconfigure themselves accordingly. Most of the default plugins interpret the values as follows:

- Setting the option to `false` disables the parser.
- Setting the option to `true` or leaving it `undefined` enables the parser.

However, parser plugins may accept more than just a boolean. For example, the internal `NodeStuffPlugin` can accept an object instead of `true` to add additional options for a particular Rule.

Examples (parser options by the default plugins):

```
parser: {
  amd: false, // disable AMD
  commonjs: false, // disable CommonJS
```

```
  system: false, // disable SystemJS
  harmony: false, // disable ES2015 Harmony import/export
  requireInclude: false, // disable require.include
  requireEnsure: false, // disable require.ensure
  requireContext: false, // disable require.context
  browserify: false, // disable special handling of Browserify bundles
  requireJs: false, // disable requirejs.*
  node: false, // disable __dirname, __filename, module, require.extensions, require.main, etc.
  node: {...} // reconfigure [node](/configuration/node) layer on module level
}
```

`Rule.resource`

A `Condition` matched with the resource. You can either supply a `Rule.resource` option or use the shortcut options `Rule.test`, `Rule.exclude`, and `Rule.include`. See details in `Rule` conditions.

`Rule.resourceQuery`

A `Condition` matched with the resource query. This option is used to test against the query section of a request string (i.e. from the question mark onwards). If you were to `import Foo from './foo.css?inline'`, the following condition would match:

```
{
  test: /.css$/,
  resourceQuery: /inline/,
  use: 'url-loader'
}
```

`Rule.rules`

An array of `Rules` that is also used when the Rule matches.

`Rule.test`

`Rule.test` is a shortcut to `Rule.resource.test`. If you supply a `Rule.test` option, you cannot also supply a `Rule.resource`. See `Rule.resource` and `Condition.test` for details.

`Rule.use`

A list of UseEntries which are applied to modules. Each entry specifies a loader to be used.

Passing a string (i.e. `use: [ "style-loader" ]`) is a shortcut to the loader property (i.e. `use: [ { loader: "style-loader "} ]`).

Loaders can be chained by passing multiple loaders, which will be applied from right to left (last to first configured).

```
use: [
  'style-loader',
  {
    loader: 'css-loader',
    options: {
      importLoaders: 1
    }
  },
  {
```

```
    loader: 'less-loader',
    options: {
      noIeCompat: true
    }
  }
]
```

See UseEntry for details.

Condition

Conditions can be one of these:

- A string: To match the input must start with the provided string. I. e. an absolute directory path, or absolute path to the file.
- A RegExp: It's tested with the input.
- A function: It's called with the input and must return a truthy value to match.
- An array of Conditions: At least one of the Conditions must match.
- An object: All properties must match. Each property has a defined behavior.

`{ test: Condition }` : The Condition must match. The convention is to provide a RegExp or array of RegExps here, but it's not enforced.

`{ include: Condition }` : The Condition must match. The convention is to provide a string or array of strings here, but it's not enforced.

`{ exclude: Condition }` : The Condition must NOT match. The convention is to provide a string or array of strings here, but it's not enforced.

`{ and: [Condition] }` : All Conditions must match.

`{ or: [Condition] }` : Any Condition must match.

`{ not: [Condition] }` : All Conditions must NOT match.

Example:

```
{
  test: /\.css$/,
  include: [
    path.resolve(__dirname, "app/styles"),
    path.resolve(__dirname, "vendor/styles")
  ]
}
```

UseEntry

`object`

It must have a `loader` property being a string. It is resolved relative to the configuration `context` with the loader resolving options (resolveLoader).

It can have an `options` property being a string or object. This value is passed to the loader, which should interpret it as loader options.

For compatibility a `query` property is also possible, which is an alias for the `options` property. Use the `options` property instead.

Example:

```
{
  loader: "css-loader",
  options: {
    modules: true
  }
}
```

Note that webpack needs to generate a unique module identifier from the resource and all loaders including options. It tries to do this with a `JSON.stringify` of the options object. This is fine in 99.9% of cases, but may be not unique if you apply the same loaders with different options to the resource and the options have some stringified values.

It also breaks if the options object cannot be stringified (i.e. circular JSON). Because of this you can have a `ident` property in the options object which is used as unique identifier.

Module Contexts

> Avoid using these options as they are **deprecated** and will soon be removed.

These options describe the default settings for the context created when a dynamic dependency is encountered.

Example for an `unknown` dynamic dependency: `require`.

Example for an `expr` dynamic dependency: `require(expr)`.

Example for an `wrapped` dynamic dependency: `require("./templates/" + expr)`.

Here are the available options with their defaults:

```
module: {
  exprContextCritical: true,
  exprContextRecursive: true,
  exprContextRegExp: false,
  exprContextRequest: ".",
  unknownContextCritical: true,
  unknownContextRecursive: true,
  unknownContextRegExp: false,
  unknownContextRequest: ".",
  wrappedContextCritical: false
  wrappedContextRecursive: true,
  wrappedContextRegExp: /.*/,
  strictExportPresence: false // since webpack 2.3.0
}
```

T> You can use the `ContextReplacementPlugin` to modify these values for individual dependencies. This also removes the warning.

A few use cases:

- Warn for dynamic dependencies: `wrappedContextCritical: true`.
- `require(expr)` should include the whole directory: `exprContextRegExp: /^\.\//`
- `require("./templates/" + expr)` should not include subdirectories by default: `wrappedContextRecursive: false`
- `strictExportPresence` makes missing exports an error instead of warning

These options change how modules are resolved. webpack provides reasonable defaults, but it is possible to change the resolving in detail. Have a look at Module Resolution for more explanation of how the resolver works.

`resolve`

`object`

Configure how modules are resolved. For example, when calling `import "lodash"` in ES2015, the `resolve` options can change where webpack goes to look for `"lodash"` (see `modules`).

`resolve.alias`

`object`

Create aliases to `import` or `require` certain modules more easily. For example, to alias a bunch of commonly used `src/` folders:

```
alias: {
  Utilities: path.resolve(__dirname, 'src/utilities/'),
  Templates: path.resolve(__dirname, 'src/templates/')
}
```

Now, instead of using relative paths when importing like so:

```
import Utility from '../../utilities/utility';
```

you can use the alias:

```
import Utility from 'Utilities/utility';
```

A trailing `$` can also be added to the given object's keys to signify an exact match:

```
alias: {
  xyz$: path.resolve(__dirname, 'path/to/file.js')
}
```

which would yield these results:

```
import Test1 from 'xyz'; // Exact match, so path/to/file.js is resolved and imported
import Test2 from 'xyz/file.js'; // Not an exact match, normal resolution takes place
```

The following table explains other cases:

`alias:`	`import "xyz"`	`import "xyz/file.js"`
`{}`	`/abc/node_modules/xyz/index.js`	`/abc/node_modules/xyz/file.js`
`{ xyz: "/abs/path/to/file.js" }`	`/abs/path/to/file.js`	error
`{ xyz$: "/abs/path/to/file.js" }`	`/abs/path/to/file.js`	`/abc/node_modules/xyz/file.js`
`{ xyz: "./dir/file.js" }`	`/abc/dir/file.js`	error
`{ xyz$: "./dir/file.js" }`	`/abc/dir/file.js`	`/abc/node_modules/xyz/file.js`
`{ xyz: "/some/dir" }`	`/some/dir/index.js`	`/some/dir/file.js`
`{ xyz$: "/some/dir" }`	`/some/dir/index.js`	`/abc/node_modules/xyz/file.js`

`{ xyz: "./dir" }`	`/abc/dir/index.js`	`/abc/dir/file.js`
`{ xyz: "modu" }`	`/abc/node_modules/modu/index.js`	`/abc/node_modules/modu/file.js`
`{ xyz$: "modu" }`	`/abc/node_modules/modu/index.js`	`/abc/node_modules/xyz/file.js`
`{ xyz: "modu/some/file.js" }`	`/abc/node_modules/modu/some/file.js`	error
`{ xyz: "modu/dir" }`	`/abc/node_modules/modu/dir/index.js`	`/abc/node_modules/dir/file.js`
`{ xyz: "xyz/dir" }`	`/abc/node_modules/xyz/dir/index.js`	`/abc/node_modules/xyz/dir/file.js`
`{ xyz$: "xyz/dir" }`	`/abc/node_modules/xyz/dir/index.js`	`/abc/node_modules/xyz/file.js`

`index.js` may resolve to another file if defined in the `package.json`.

`/abc/node_modules` may resolve in `/node_modules` too.

`resolve.aliasFields`

`string`

Specify a field, such as `browser`, to be parsed according to this specification. Default:

```
aliasFields: ["browser"]
```

`resolve.cacheWithContext`

`boolean` (since webpack 3.1.0)

If unsafe cache is enabled, includes `request.context` in the cache key. This option is taken into account by the `enhanced-resolve` module. Since webpack 3.1.0 context in resolve caching is ignored when resolve or resolveLoader plugins are provided. This addresses a performance regression.

`resolve.descriptionFiles`

`array`

The JSON files to use for descriptions. Default:

```
descriptionFiles: ["package.json"]
```

`resolve.enforceExtension`

`boolean`

If `true`, it will not allow extension-less files. So by default `require('./foo')` works if `./foo` has a `.js` extension, but with this enabled only `require('./foo.js')` will work. Default:

```
enforceExtension: false
```

`resolve.enforceModuleExtension`

`boolean`

Whether to require to use an extension for modules (e.g. loaders). Default:

```
enforceModuleExtension: false
```

`resolve.extensions`

`array`

Automatically resolve certain extensions. This defaults to:

```
extensions: [".js", ".json"]
```

which is what enables users to leave off the extension when importing:

```
import File from '../path/to/file'
```

W> Using this will **override the default array**, meaning that webpack will no longer try to resolve modules using the default extensions. For modules that are imported with their extension, e.g. `import SomeFile from "./somefile.ext"`, to be properly resolved, a string containing "*****" must be included in the array.

`resolve.mainFields`

`array`

When importing from an npm package, e.g. `import * as D3 from "d3"`, this option will determine which fields in its `package.json` are checked. The default values will vary based upon the `target` specified in your webpack configuration.

When the `target` property is set to `webworker`, `web`, or left unspecified:

```
mainFields: ["browser", "module", "main"]
```

For any other target (including `node`):

```
mainFields: ["module", "main"]
```

For example, the `package.json` of D3 contains these fields:

```
{
  ...
  main: 'build/d3.Node.js',
  browser: 'build/d3.js',
  module: 'index',
  ...
}
```

This means that when we `import * as D3 from "d3"` this will really resolve to the file in the `browser` property. The `browser` property takes precedence here because it's the first item in `mainFields`. Meanwhile, a Node.js application bundled by webpack will resolve by default to the file in the `module` field.

`resolve.mainFiles`

`array`

The filename to be used while resolving directories. Default:

```
mainFiles: ["index"]
```

`resolve.modules`

`array`

Tell webpack what directories should be searched when resolving modules.

Absolute and relative paths can both be used, but be aware that they will behave a bit differently.

A relative path will be scanned similarly to how Node scans for `node_modules`, by looking through the current directory as well as its ancestors (i.e. `./node_modules`, `../node_modules`, and on).

With an absolute path, it will only search in the given directory.

`resolve.modules` defaults to:

```
modules: ["node_modules"]
```

If you want to add a directory to search in that takes precedence over `node_modules/`:

```
modules: [path.resolve(__dirname, "src"), "node_modules"]
```

`resolve.unsafeCache`

`regex` `array` `boolean`

Enable aggressive, but **unsafe**, caching of modules. Passing `true` will cache everything. Default:

```
unsafeCache: true
```

A regular expression, or an array of regular expressions, can be used to test file paths and only cache certain modules. For example, to only cache utilities:

```
unsafeCache: /src\/utilities/
```

W> Changes to cached paths may cause failure in rare cases.

`resolve.plugins`

A list of additional resolve plugins which should be applied. It allows plugins such as `DirectoryNamedWebpackPlugin`.

```
plugins: [
  new DirectoryNamedWebpackPlugin()
]
```

`resolve.symlinks`

`boolean`

Whether to resolve symlinks to their symlinked location.

When enabled, symlinked resources are resolved to their *real* path, not their symlinked location. Note that this may cause module resolution to fail when using tools that symlink packages (like `npm link`).

`resolve.symlinks` defaults to:

```
symlinks: true
```

`resolve.cachePredicate`

`function`

A function which decides whether a request should be cached or not. An object is passed to the function with `path` and `request` properties. Default:

```
cachePredicate: function() { return true }
```

`resolveLoader`

`object`

This set of options is identical to the `resolve` property set above, but is used only to resolve webpack's loader packages. Default:

```
{
  modules: [ 'node_modules' ],
  extensions: [ '.js', '.json' ],
  mainFields: [ 'loader', 'main' ]
}
```

T> Note that you can use alias here and other features familiar from resolve. For example `{ txt: 'raw-loader' }` would shim `txt!templates/demo.txt` to use `raw-loader`.

`resolveLoader.moduleExtensions`

`array`

The extensions/suffixes which that are used when resolving loaders. Since version two, we strongly recommend using the full name, e.g. `example-loader`, as much as possible for clarity. However, if you really wanted to exclude the `-loader` bit, i.e. just use `example`, you can use this option to do so:

```
moduleExtensions: [ '-loader' ]
```

The `plugins` option is used to customize the webpack build process in a variety of ways. webpack comes with a variety built-in plugins available under `webpack.[plugin-name]`. See this page for a list of plugins and documentation but note that there are a lot more out in the community.

T> Note: This page only discusses using plugins, however if you are interested in writing your own please visit Writing a Plugin.

`plugins`

`array`

A list of webpack plugins. For example, when multiple bundles share some of the same dependencies, the `CommonsChunkPlugin` could be useful to extract those dependencies into a shared bundle to avoid duplication. This could be added like so:

```
plugins: [
  new webpack.optimize.CommonsChunkPlugin({
    ...
  })
]
```

A more complex example, using multiple plugins, might look something like this:

```
var webpack = require('webpack');
// importing plugins that do not come by default in webpack
var ExtractTextPlugin = require('extract-text-webpack-plugin');
var DashboardPlugin = require('webpack-dashboard/plugin');

// adding plugins to your configuration
plugins: [
  // build optimization plugins
  new webpack.optimize.CommonsChunkPlugin({
    name: 'vendor',
    filename: 'vendor-[hash].min.js',
  }),
  new webpack.optimize.UglifyJsPlugin({
    compress: {
      warnings: false,
      drop_console: false,
    }
  }),
  new ExtractTextPlugin({
    filename: 'build.min.css',
    allChunks: true,
  }),
  new webpack.IgnorePlugin(/^\.\/locale$/, /moment$/),
  // compile time plugins
  new webpack.DefinePlugin({
    'process.env.NODE_ENV': '"production"',
  }),
  // webpack-dev-server enhancement plugins
  new DashboardPlugin(),
  new webpack.HotModuleReplacementPlugin(),
]
```

webpack-dev-server can be used to quickly develop an application. See the "How to Develop?" to get started.

This page describes the options that affect the behavior of webpack-dev-server (short: dev-server).

T> Options that are compatible with webpack-dev-middleware have next to them.

devServer

`object`

This set of options is picked up by webpack-dev-server and can be used to change its behavior in various ways. Here's a simple example that gzips and serves everything from our `dist/` directory:

```
devServer: {
  contentBase: path.join(__dirname, "dist"),
  compress: true,
  port: 9000
}
```

When the server is started, there will be a message prior to the list of resolved modules:

```
http://localhost:9000/
webpack output is served from /build/
Content not from webpack is served from /path/to/dist/
```

that will give some background on where the server is located and what it's serving.

If you're using dev-server through the Node.js API, the options in `devServer` will be ignored. Pass the options as a second parameter instead: `new WebpackDevServer(compiler, {...})`. See here for an example of how to use webpack-dev-server through the Node.js API.

W> Be aware that when exporting multiple configurations only the `devServer` options for the first configuration will be taken into account and used for all the configurations in the array.

T> If you're having trouble, navigating to the `/webpack-dev-server` route will show where files are served. For example, `http://localhost:9000/webpack-dev-server`.

devServer.after

`function`

Provides the ability to execute custom middleware after all other middleware internally within the server.

```
after(app){
  // do fancy stuff
}
```

devServer.allowedHosts

`array`

This option allows you to whitelist services that are allowed to access the dev server.

```
allowedHosts: [
  'host.com',
  'subdomain.host.com',
  'subdomain2.host.com',
```

```
  'host2.com'
]
```

Mimicking django's `ALLOWED_HOSTS`, a value beginning with `.` can be used as a subdomain wildcard. `.host.com` will match `host.com`, `www.host.com`, and any other subdomain of `host.com`.

```
// this achieves the same effect as the first example
// with the bonus of not having to update your config
// if new subdomains need to access the dev server
allowedHosts: [
    '.host.com',
    'host2.com'
]
```

To use this option with the CLI pass the `--allowed-hosts` option a comma-delimited string.

```
webpack-dev-server --entry /entry/file --output-path /output/path --allowed-hosts .host.com,host2.com
```

devServer.before

`function`

Provides the ability to execute custom middleware prior to all other middleware internally within the server. This could be used to define custom handlers, for example:

```
before(app){
  app.get('/some/path', function(req, res) {
    res.json({ custom: 'response' });
  });
}
```

devServer.bonjour

This option broadcasts the server via ZeroConf networking on start

```
bonjour: true
```

Usage via the CLI

```
webpack-dev-server --bonjour
```

devServer.clientLogLevel

`string`

When using *inline mode*, the console in your DevTools will show you messages e.g. before reloading, before an error or when Hot Module Replacement is enabled. This may be too verbose.

You can prevent all these messages from showing, by using this option:

```
clientLogLevel: "none"
```

Usage via the CLI

```
webpack-dev-server --client-log-level none
```

Possible values are `none`, `error`, `warning` or `info` (default).

`devServer.color` - CLI only

`boolean`

Enables/Disables colors on the console.

```
webpack-dev-server --color
```

`devServer.compress`

`boolean`

Enable gzip compression for everything served:

```
compress: true
```

Usage via the CLI

```
webpack-dev-server --compress
```

`devServer.contentBase`

`boolean` `string` `array`

Tell the server where to serve content from. This is only necessary if you want to serve static files. `devServer.publicPath` will be used to determine where the bundles should be served from, and takes precedence.

By default it will use your current working directory to serve content, but you can modify this to another directory:

```
contentBase: path.join(__dirname, "public")
```

Note that it is recommended to use an absolute path.

It is also possible to serve from multiple directories:

```
contentBase: [path.join(__dirname, "public"), path.join(__dirname, "assets")]
```

To disable `contentBase`:

```
contentBase: false
```

Usage via the CLI

```
webpack-dev-server --content-base /path/to/content/dir
```

`devServer.disableHostCheck`

`boolean`

When set to true this option bypasses host checking. THIS IS NOT RECOMMENDED as apps that do not check the host are vulnerable to DNS rebinding attacks.

```
disableHostCheck: true
```

Usage via the CLI

```
webpack-dev-server --disable-host-check
```

`devServer.filename`

`string`

This option lets you reduce the compilations in **lazy mode**. By default in **lazy mode**, every request results in a new compilation. With `filename`, it's possible to only compile when a certain file is requested.

If `output.filename` is set to `bundle.js` and `filename` is used like this:

```
lazy: true,
filename: "bundle.js"
```

It will now only compile the bundle when `/bundle.js` is requested.

T> `filename` has no effect when used without **lazy mode**.

`devServer.headers`

`object`

Adds headers to all responses:

```
headers: {
  "X-Custom-Foo": "bar"
}
```

`devServer.historyApiFallback`

`boolean` `object`

When using the HTML5 History API, the `index.html` page will likely have to be served in place of any `404` responses. Enable this by passing:

```
historyApiFallback: true
```

By passing an object this behavior can be controlled further using options like `rewrites`:

```
historyApiFallback: {
  rewrites: [
    { from: /^\/$/, to: '/views/landing.html' },
    { from: /^\/subpage/, to: '/views/subpage.html' },
    { from: /./, to: '/views/404.html' }
  ]
```

```
}
```

When using dots in your path (common with Angular), you may need to use the `disableDotRule`:

```
historyApiFallback: {
  disableDotRule: true
}
```

Usage via the CLI

```
webpack-dev-server --history-api-fallback
```

For more options and information, see the connect-history-api-fallback documentation.

`devServer.host`

`string`

Specify a host to use. By default this is `localhost`. If you want your server to be accessible externally, specify it like this:

```
host: "0.0.0.0"
```

Usage via the CLI

```
webpack-dev-server --host 0.0.0.0
```

`devServer.hot`

`boolean`

Enable webpack's Hot Module Replacement feature:

```
hot: true
```

T> Note that `webpack.HotModuleReplacementPlugin` is required to fully enable HMR. If `webpack` or `webpack-dev-server` are launched with the `--hot` option, this plugin will be added automatically, so you may not need to add this to your `webpack.config.js`. See the HMR concepts page for more information.

`devServer.hotOnly`

`boolean`

Enables Hot Module Replacement (see `devServer.hot`) without page refresh as fallback in case of build failures.

```
hotOnly: true
```

Usage via the CLI

```
webpack-dev-server --hot-only
```

`devServer.https`

`boolean` `object`

By default dev-server will be served over HTTP. It can optionally be served over HTTP/2 with HTTPS:

```
https: true
```

With the above setting a self-signed certificate is used, but you can provide your own:

```
https: {
  key: fs.readFileSync("/path/to/server.key"),
  cert: fs.readFileSync("/path/to/server.crt"),
  ca: fs.readFileSync("/path/to/ca.pem"),
}
```

This object is passed straight to Node.js HTTPS module, so see the HTTPS documentation for more information.

Usage via the CLI

```
webpack-dev-server --https
```

To pass your own certificate via the CLI use the following options

```
webpack-dev-server --https --key /path/to/server.key --cert /path/to/server.crt --cacert /path/to/ca.pem
```

`devServer.index`

`string`

The filename that is considered the index file.

```
index: 'index.htm'
```

`devServer.info` - CLI only

`boolean`

Output cli information. It is enabled by default.

```
webpack-dev-server --info=false
```

`devServer.inline`

`boolean`

Toggle between the dev-server's two different modes. By default the application will be served with *inline mode* enabled. This means that a script will be inserted in your bundle to take care of live reloading, and build messages will appear in the browser console.

It is also possible to use **iframe mode**, which uses an `<iframe>` under a notification bar with messages about the build. To switch to **iframe mode**:

```
inline: false
```

Usage via the CLI

```
webpack-dev-server --inline=false
```

T> Inline mode is recommended for Hot Module Replacement as it includes an HMR trigger from the websocket. Polling mode can be used as an alternative, but requires an additional entry point, `'webpack/hot/poll?1000'`.

`devServer.lazy`

`boolean`

When `lazy` is enabled, the dev-server will only compile the bundle when it gets requested. This means that webpack will not watch any file changes. We call this **lazy mode**.

```
lazy: true
```

Usage via the CLI

```
webpack-dev-server --lazy
```

T> `watchOptions` will have no effect when used with **lazy mode**.

T> If you use the CLI, make sure **inline mode** is disabled.

`devServer.noInfo`

`boolean`

With `noInfo` enabled, messages like the webpack bundle information that is shown when starting up and after each save, will be hidden. Errors and warnings will still be shown.

```
noInfo: true
```

`devServer.open`

`boolean`

When `open` is enabled, the dev server will open the browser.

```
open: true
```

Usage via the CLI

```
webpack-dev-server --open
```

If no browser is provided (as shown above), your default browser will be used. To specify a different browser, just pass its name:

```
webpack-dev-server --open 'Google Chrome'
```

devServer.openPage

`string`

Specify a page to navigate to when opening the browser.

```
openPage: '/different/page'
```

Usage via the CLI

```
webpack-dev-server --open-page "/different/page"
```

devServer.overlay

`boolean` `object`

Shows a full-screen overlay in the browser when there are compiler errors or warnings. Disabled by default. If you want to show only compiler errors:

```
overlay: true
```

If you want to show warnings as well as errors:

```
overlay: {
  warnings: true,
  errors: true
}
```

devServer.pfx

`string`

When used via the CLI, a path to an SSL .pfx file. If used in options, it should be the bytestream of the .pfx file.

```
pfx: '/path/to/file.pfx'
```

Usage via the CLI

```
webpack-dev-server --pfx /path/to/file.pfx
```

devServer.pfxPassphrase

`string`

The passphrase to a SSL PFX file.

```
pfxPassphrase: 'passphrase'
```

Usage via the CLI

```
webpack-dev-server --pfx-passphrase passphrase
```

devServer.port

`number`

Specify a port number to listen for requests on:

```
port: 8080
```

Usage via the CLI

```
webpack-dev-server --port 8080
```

devServer.proxy

`object`

Proxying some URLs can be useful when you have a separate API backend development server and you want to send API requests on the same domain.

The dev-server makes use of the powerful http-proxy-middleware package. Checkout its documentation for more advanced usages.

With a backend on `localhost:3000`, you can use this to enable proxying:

```
proxy: {
  "/api": "http://localhost:3000"
}
```

A request to `/api/users` will now proxy the request to `http://localhost:3000/api/users`.

If you don't want `/api` to be passed along, we need to rewrite the path:

```
proxy: {
  "/api": {
    target: "http://localhost:3000",
    pathRewrite: {"^/api" : ""}
  }
}
```

A backend server running on HTTPS with an invalid certificate will not be accepted by default. If you want to, modify your config like this:

```
proxy: {
  "/api": {
    target: "https://other-server.example.com",
    secure: false
  }
}
```

Sometimes you don't want to proxy everything. It is possible to bypass the proxy based on the return value of a function.

In the function you get access to the request, response and proxy options. It must return either `false` or a path that will be served instead of continuing to proxy the request.

E.g. for a browser request, you want to serve a HTML page, but for an API request you want to proxy it. You could do something like this:

```
proxy: {
  "/api": {
    target: "http://localhost:3000",
    bypass: function(req, res, proxyOptions) {
      if (req.headers.accept.indexOf("html") !== -1) {
        console.log("Skipping proxy for browser request.");
        return "/index.html";
      }
    }
  }
}
```

If you want to proxy multiple, specific paths to the same target, you can use an array of one or more objects with a `context` property:

```
proxy: [{
  context: ["/auth", "/api"],
  target: "http://localhost:3000",
}]
```

`devServer.progress` - CLI only

`boolean`

Output running progress to console.

```
webpack-dev-server --progress
```

`devServer.public`

`string`

When using *inline mode* and you're proxying dev-server, the inline client script does not always know where to connect to. It will try to guess the URL of the server based on `window.location`, but if that fails you'll need to use this.

For example, the dev-server is proxied by nginx, and available on `myapp.test`:

```
public: "myapp.test:80"
```

Usage via the CLI

```
webpack-dev-server --public myapp.test:80
```

`devServer.publicPath`

`string`

The bundled files will be available in the browser under this path.

Imagine that the server is running under `http://localhost:8080` and `output.filename` is set to `bundle.js`. By default the `publicPath` is `"/"`, so your bundle is available as `http://localhost:8080/bundle.js`.

The `publicPath` can be changed so the bundle is put in a directory:

```
publicPath: "/assets/"
```

The bundle will now be available as `http://localhost:8080/assets/bundle.js`.

T> Make sure `publicPath` always starts and ends with a forward slash.

It is also possible to use a full URL. This is necessary for Hot Module Replacement.

```
publicPath: "http://localhost:8080/assets/"
```

The bundle will also be available as `http://localhost:8080/assets/bundle.js`.

T> It is recommended that `devServer.publicPath` is the same as `output.publicPath`.

`devServer.quiet`

`boolean`

With `quiet` enabled, nothing except the initial startup information will be written to the console. This also means that errors or warnings from webpack are not visible.

```
quiet: true
```

Usage via the CLI

```
webpack-dev-server --quiet
```

`devServer.setup`

`function`

W> This option is **deprecated** in favor of `before` and will be removed in v3.0.0.

Here you can access the Express app object and add your own custom middleware to it. For example, to define custom handlers for some paths:

```
setup(app){
  app.get('/some/path', function(req, res) {
    res.json({ custom: 'response' });
  });
}
```

`devServer.socket`

`string`

The Unix socket to listen to (instead of a host).

```
socket: 'socket'
```

Usage via the CLI

```
webpack-dev-server --socket socket
```

`devServer.staticOptions`

It is possible to configure advanced options for serving static files from `contentBase`. See the Express documentation for the possible options. An example:

```
staticOptions: {
  redirect: false
}
```

T> This only works when using `contentBase` as a `string`.

`devServer.stats`

`string` `object`

This option lets you precisely control what bundle information gets displayed. This can be a nice middle ground if you want some bundle information, but not all of it.

To show only errors in your bundle:

```
stats: "errors-only"
```

For more information, see the **stats documentation**.

T> This option has no effect when used with `quiet` or `noInfo`.

`devServer.stdin` - CLI only

`boolean`

This option closes the server when stdin ends.

```
webpack-dev-server --stdin
```

`devServer.useLocalIp`

`boolean`

This option lets the browser open with your local IP.

```
useLocalIp: true
```

Usage via the CLI

```
webpack-dev-server --useLocalIp
```

`devServer.watchContentBase`

`boolean`

Tell the server to watch the files served by the `devServer.contentBase` option. File changes will trigger a full page reload.

```
watchContentBase: true
```

It is disabled by default.

Usage via the CLI

```
webpack-dev-server --watch-content-base
```

`devServer.watchOptions`

`object`

Control options related to watching the files.

webpack uses the file system to get notified of file changes. In some cases this does not work. For example, when using Network File System (NFS). Vagrant also has a lot of problems with this. In these cases, use polling:

```
watchOptions: {
  poll: true
}
```

If this is too heavy on the file system, you can change this to an integer to set the interval in milliseconds.

See WatchOptions for more options.

This option controls if and how source maps are generated.

Use the `SourceMapDevToolPlugin` for a more fine grained configuration. See the `source-map-loader` to deal with existing source maps.

`devtool`

`string` `false`

Choose a style of source mapping to enhance the debugging process. These values can affect build and rebuild speed dramatically.

T> The webpack repository contains an example showing the effect of all `devtool` variants. Those examples will likely help you to understand the differences.

T> Instead of using the `devtool` option you can also use `SourceMapDevToolPlugin` / `EvalSourceMapDevToolPlugin` directly as it has more options. Never use both the `devtool` option and plugin together. The `devtool` option adds the plugin internally so you would end up with the plugin applied twice.

devtool	build	rebuild	production	quality
(none)	+++	+++	yes	bundled code
eval	+++	+++	no	generated code
cheap-eval-source-map	+	++	no	transformed code (lines only)
cheap-module-eval-source-map	o	++	no	original source (lines only)
eval-source-map	--	+	no	original source
cheap-source-map	+	o	no	transformed code (lines only)
cheap-module-source-map	o	-	no	original source (lines only)
inline-cheap-source-map	+	o	no	transformed code (lines only)
inline-cheap-module-source-map	o	-	no	original source (lines only)
source-map	--	--	yes	original source
inline-source-map	--	--	no	original source
hidden-source-map	--	--	yes	original source
nosources-source-map	--	--	yes	without source content

T> `+++` super fast, `++` fast, `+` pretty fast, `o` medium, `-` pretty slow, `--` slow

Some of these values are suited for development and some for production. For development you typically want fast Source Maps at the cost of bundle size, but for production you want separate Source Maps that are accurate and support minimizing.

W> There are some issues with Source Maps in Chrome. We need your help!.

T> See `output.sourceMapFilename` to customize the filenames of generated Source Maps.

Qualities

`bundled code` - You see all generated code as a big blob of code. You don't see modules separated from each other.

`generated code` - You see each module separated from each other, annotated with module names. You see the code generated by webpack. Example: Instead of `import {test} from "module"; test();` you see something like `var module__WEBPACK_IMPORTED_MODULE_1__ = __webpack_require__(42); module__WEBPACK_IMPORTED_MODULE_1__.a();`.

`transformed code` - You see each module separated from each other, annotated with module names. You see the code before webpack transforms it, but after Loaders transpile it. Example: Instead of `import {test} from "module"; class A extends test {}` you see something like `import {test} from "module"; var A = function(_test) { ... }(test);`

`original source` - You see each module separated from each other, annotated with module names. You see the code before transpilation, as you authored it. This depends on Loader support.

`without source content` - Contents for the sources are not included in the Source Maps. Browsers usually try to load the source from the webserver or filesystem. You have to make sure to set `output.devtoolModuleFilenameTemplate` correctly to match source urls.

`(lines only)` - Source Maps are simplified to a single mapping per line. This usually means a single mapping per statement (assuming you author is this way). This prevents you from debugging execution on statement level and from settings breakpoints on columns of a line. Combining with minimizing is not possible as minimizers usually only emit a single line.

Development

The following options are ideal for development:

`eval` - Each module is executed with `eval()` and `//@ sourceURL`. This is pretty fast. The main disadvantage is that it doesn't display line numbers correctly since it gets mapped to transpiled code instead of the original code (No Source Maps from Loaders).

`eval-source-map` - Each module is executed with `eval()` and a SourceMap is added as a DataUrl to the `eval()`. Initially it is slow, but it provides fast rebuild speed and yields real files. Line numbers are correctly mapped since it gets mapped to the original code. It yields the best quality SourceMaps for development.

`cheap-eval-source-map` - Similar to `eval-source-map`, each module is executed with `eval()`. It is "cheap" because it doesn't have column mappings, it only maps line numbers. It ignores SourceMaps from Loaders and only display transpiled code similar to the `eval` devtool.

`cheap-module-eval-source-map` - Similar to `cheap-eval-source-map`, however, in this case Source Maps from Loaders are processed for better results. However Loader Source Maps are simplified to a single mapping per line.

Special cases

The following options are not ideal for development nor production. They are needed for some special cases, i. e. for some 3rd party tools.

`inline-source-map` - A SourceMap is added as a DataUrl to the bundle.

`cheap-source-map` - A SourceMap without column-mappings ignoring loader Source Maps.

`inline-cheap-source-map` - Similar to `cheap-source-map` but SourceMap is added as a DataUrl to the bundle.

`cheap-module-source-map` - A SourceMap without column-mappings that simplifies loader Source Maps to a single mapping per line.

`inline-cheap-module-source-map` - Similar to `cheap-module-source-map` but SourceMap is added as a DataUrl to the bundle.

Production

These options are typically used in production:

`(none)` (Omit the `devtool` option) - No SourceMap is emitted. This is a good option to start with.

`source-map` - A full SourceMap is emitted as a separate file. It adds a reference comment to the bundle so development tools know where to find it.

W> You should configure your server to disallow access to the Source Map file for normal users!

`hidden-source-map` - Same as `source-map`, but doesn't add a reference comment to the bundle. Useful if you only want SourceMaps to map error stack traces from error reports, but don't want to expose your SourceMap for the browser development tools.

W> You should not deploy the Source Map file to the webserver. Instead only use it for error report tooling.

`nosources-source-map` - A SourceMap is created without the `sourcesContent` in it. It can be used to map stack traces on the client without exposing all of the source code. You can deploy the Source Map file to the webserver.

W> It still exposes filenames and structure for decompiling, but it doesn't expose the original code.

T> When using the `uglifyjs-webpack-plugin` you must provide the `sourceMap: true` option to enable SourceMap support.

webpack can compile for multiple environments or *targets*. To understand what a `target` is in detail, read through the targets concept page.

`target`

`string | function(compiler)`

Instructs webpack to target a specific environment.

`string`

The following string values are supported via `WebpackOptionsApply`:

Option	Description
`async-node`	Compile for usage in a Node.js-like environment (uses `fs` and `vm` to load chunks asynchronously)
~~`atom`~~	Alias for `electron-main`
~~`electron`~~	Alias for `electron-main`
`electron-main`	Compile for Electron for main process.
`electron-renderer`	Compile for Electron for renderer process, providing a target using `JsonpTemplatePlugin`, `FunctionModulePlugin` for browser environments and `NodeTargetPlugin` and `ExternalsPlugin` for CommonJS and Electron built-in modules.
`node`	Compile for usage in a Node.js-like environment (uses Node.js `require` to load chunks)
`node-webkit`	Compile for usage in WebKit and uses JSONP for chunk loading. Allows importing of built-in Node.js modules and `nw.gui` (experimental)
`web`	Compile for usage in a browser-like environment **(default)**
`webworker`	Compile as WebWorker

For example, when the `target` is set to `"electron"`, webpack includes multiple electron specific variables. For more information on which templates and externals are used, you can refer to webpack's source code.

`function`

If a function is passed, then it will be called with the compiler as a parameter. Set it to a function if none of the predefined targets from the list above meet your needs.

For example, if you don't want any of the plugins they applied:

```
const options = {
  target: () => undefined
};
```

Or you can apply specific plugins you want:

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

const options = {
  target: (compiler) => {
    compiler.apply(
      new webpack.JsonpTemplatePlugin(options.output),
      new webpack.LoaderTargetPlugin("web")
    );
```

```
  }
};
```

webpack can watch files and recompile whenever they change. This page explains how to enable this and a couple of tweaks you can make if watching does not work properly for you.

`watch`

`boolean`

Turn on watch mode. This means that after the initial build, webpack will continue to watch for changes in any of the resolved files. Watch mode is turned off by default:

```
watch: false
```

T> In webpack-dev-server and webpack-dev-middleware watch mode is enabled by default.

`watchOptions`

`object`

A set of options used to customize watch mode:

```
watchOptions: {
  aggregateTimeout: 300,
  poll: 1000
}
```

`watchOptions.aggregateTimeout`

`number`

Add a delay before rebuilding once the first file changed. This allows webpack to aggregate any other changes made during this time period into one rebuild. Pass a value in milliseconds:

```
aggregateTimeout: 300 // The default
```

`watchOptions.ignored`

For some systems, watching many file systems can result in a lot of CPU or memory usage. It is possible to exclude a huge folder like `node_modules`:

```
ignored: /node_modules/
```

It is also possible to use anymatch patterns:

```
ignored: "files/**/*.js"
```

`watchOptions.poll`

`boolean` `number`

Turn on polling by passing `true`, or specifying a poll interval in milliseconds:

```
poll: 1000 // Check for changes every second
```

T> If watching does not work for you, try out this option. Watching does not work with NFS and machines in VirtualBox.

`info-verbosity`

`string` : `none` `info` `verbose`

Controls verbosity of the lifecycle messaging, e.g. the `Started watching files...` log. Setting `info-verbosity` to `verbose` will also message to console at the beginning and the end of incremental build. `info-verbosity` is set to `info` by default.

```
webpack --watch --info-verbosity verbose
```

Troubleshooting

If you are experiencing any issues, please see the following notes. There are a variety of reasons why webpack might miss a file change.

Changes Seen But Not Processed

Verify that webpack is not being notified of changes by running webpack with the --progress flag. If progress shows on save but no files are outputted, it is likely a configuration issue, not a file watching issue.

```
webpack --watch --progress
```

Not Enough Watchers

Verify that you have enough available watchers in your system. If this value is too low, the file watcher in Webpack won't recognize the changes:

```
cat /proc/sys/fs/inotify/max_user_watches
```

Arch users, add `fs.inotify.max_user_watches=524288` to `/etc/sysctl.d/99-sysctl.conf` and then execute `sysctl --system`. Ubuntu users (and possibly others), execute: `echo fs.inotify.max_user_watches=524288 | sudo tee -a /etc/sysctl.conf && sudo sysctl -p`.

MacOS fsevents Bug

On MacOS, folders can get corrupted in certain scenarios. See this article.

Windows Paths

Because webpack expects absolute paths for many config options such as `__dirname + "/app/folder"` the Windows `\` path separator can break some functionality.

Use the correct separators. I.e. `path.resolve(__dirname, "app/folder")` or `path.join(__dirname, "app", "folder")`.

Vim

On some machines Vim is preconfigured with the backupcopy option set to `auto`. This could potentially cause problems with the system's file watching mechanism. Switching this option to `yes` will make sure a copy of the file is made and the original one overwritten on save.

```
:set backupcopy=yes
```

Saving in WebStorm

When using the JetBrains WebStorm IDE, you may find that saving changed files does not trigger the watcher as you might expect. Try disabling the `safe write` option in the settings, which determines whether files are saved to a temporary location first before the originals are overwritten: uncheck `File > Settings... > System Settings > Use "safe write" (save changes to a temporary file first)`.

The `externals` configuration option provides a way of excluding dependencies from the output bundles. Instead, the created bundle relies on that dependency to be present in the consumer's environment. This feature is typically most useful to **library developers**, however there are a variety of applications for it.

T> **consumer** here is any end user application that includes the library that you have bundled using webpack.

`externals`

`string` `array` `object` `function` `regex`

Prevent bundling of certain `import`ed packages and instead retrieve these *external dependencies* at runtime.

For example, to include jQuery from a CDN instead of bundling it:

index.html

```
<script
  src="https://code.jquery.com/jquery-3.1.0.js"
  integrity="sha256-slogkvB1K3VOkzAI8QITxV3VzpOnkeNVsKvtkYLMjfk="
  crossorigin="anonymous">
</script>
```

webpack.config.js

```
externals: {
  jquery: 'jQuery'
}
```

This leaves any dependent modules unchanged, i.e. the code shown below will still work:

```
import $ from 'jquery';

$('.my-element').animate(...);
```

The bundle with external dependencies can be used in various module contexts, such as CommonJS, AMD, global and ES2015 modules. The external library may be available in any of these forms:

- **root**: The library should be available as a global variable (e.g. via a script tag).
- **commonjs**: The library should be available as a CommonJS module.
- **commonjs2**: Similar to the above but where the export is `module.exports.default`.
- **amd**: Similar to `commonjs` but using AMD module system.

The following syntaxes are accepted...

string

See the example above. The property name `jquery` indicates that the module `jquery` in `import $ from 'jquery'` should be excluded. In order to replace this module, the value `jQuery` will be used to retrieve a global `jQuery` variable. In other words, when a string is provided it will be treated as `root` (defined above and below).

array

```
externals: {
  subtract: ['./math', 'subtract']
}
```

`subtract: ['./math', 'subtract']` converts to a parent child construct, where `./math` is the parent module and your bundle only requires the subset under `subtract` variable.

object

```
externals : {
  react: 'react'
}

// or

externals : {
  lodash : {
    commonjs: "lodash",
    amd: "lodash",
    root: "_" // indicates global variable
  }
}

// or

externals : {
  subtract : {
    root: ["math", "subtract"]
  }
}
```

This syntax is used to describe all the possible ways that an external library can be available. `lodash` here is available as `lodash` under AMD and CommonJS module systems but available as `_` in a global variable form. `subtract` here is available via the property `subtract` under the global `math` object (e.g. `window['math']['subtract']`).

function

It might be useful to define your own function to control the behavior of what you want to externalize from webpack. webpack-node-externals, for example, excludes all modules from the `node_modules` directory and provides some options to, for example, whitelist packages.

It basically comes down to this:

```
externals: [
  function(context, request, callback) {
    if (/^yourregex$/.test(request)){
      return callback(null, 'commonjs ' + request);
    }
    callback();
  }
],
```

The `'commonjs ' + request` defines the type of module that needs to be externalized.

regex

Every dependency that matches the given regular expression will be excluded from the output bundles.

```
externals: /^(jquery|\$)$/i
```

In this case any dependency named `jQuery`, capitalized or not, or `$` would be externalized.

For more information on how to use this configuration, please refer to the article on how to author a library.

These options allows you to control how webpack notifies you of assets and entrypoints that exceed a specific file limit. This feature was inspired by the idea of webpack Performance Budgets.

`performance`

`object`

Configure how performance hints are shown. For example if you have an asset that is over 250kb, webpack will emit a warning notifying you of this.

`performance.hints`

`false | "error" | "warning"`

Turns hints on/off. In addition, tells webpack to throw either an error or a warning when hints are found. This property is set to `"warning"` by default.

Given an asset is created that is over 250kb:

```
performance: {
  hints: false
}
```

No hint warnings or errors are shown.

```
performance: {
  hints: "warning"
}
```

A warning will be displayed notifying you of a large asset. We recommend something like this for development environments.

```
performance: {
  hints: "error"
}
```

An error will be displayed notifying you of a large asset. We recommend using `hints: "error"` during production builds to help prevent deploying production bundles that are too large, impacting webpage performance.

`performance.maxEntrypointSize`

`int`

An entrypoint represents all assets that would be utilized during initial load time for a specific entry. This option controls when webpack should emit performance hints based on the maximum entrypoint size. The default value is `250000` (bytes).

```
performance: {
  maxEntrypointSize: 400000
}
```

`performance.maxAssetSize`

`int`

An asset is any emitted file from webpack. This option controls when webpack emits a performance hint based on individual asset size. The default value is `250000` (bytes).

```
performance: {
  maxAssetSize: 100000
}
```

`performance.assetFilter`

`Function`

This property allows webpack to control what files are used to calculate performance hints. The default function is seen below:

```
function(assetFilename) {
    return !(/\.map$/.test(assetFilename))
};
```

You can override this property by passing your own function in:

```
performance: {
  assetFilter: function(assetFilename) {
    return assetFilename.endsWith('.js');
  }
}
```

The example above will only give you performance hints based on `.js` files.

These options configure whether to polyfill or mock certain Node.js globals and modules. This allows code originally written for the Node.js environment to run in other environments like the browser.

This feature is provided by webpack's internal `NodeStuffPlugin` plugin. If the target is "web" (default) or "webworker", the `NodeSourcePlugin` plugin is also activated.

node

`object`

This is an object where each property is the name of a Node global or module and each value may be one of the following...

- `true` : Provide a polyfill.
- `"mock"` : Provide a mock that implements the expected interface but has little or no functionality.
- `"empty"` : Provide an empty object.
- `false` : Provide nothing. Code that expects this object may crash with a `ReferenceError` . Code that attempts to import the module using `require('modulename')` may trigger a `Cannot find module "modulename"` error.

W> Not every Node global supports all four options. The compiler will throw an error for property-value combinations that aren't supported (e.g. `process: 'empty'`). See the sections below for more details.

These are the defaults:

```
node: {
  console: false,
  global: true,
  process: true,
  __filename: "mock",
  __dirname: "mock",
  Buffer: true,
  setImmediate: true

  // See "Other node core libraries" for additional options.
}
```

Since webpack 3.0.0, the `node` option may be set to `false` to completely turn off the `NodeStuffPlugin` and `NodeSourcePlugin` plugins.

node.console

`boolean | "mock"`

Default: `false`

The browser provides a `console` object with a very similar interface to the Node.js `console` , so a polyfill is generally not needed.

node.process

`boolean | "mock"`

Default: `true`

node.global

`boolean`

Default: `true`

See the source for the exact behavior of this object.

`node.__filename`

`boolean | "mock"`

Default: `"mock"`

Options:

- `true` : The filename of the **input** file relative to the `context` option.
- `false` : The regular Node.js `__filename` behavior. The filename of the **output** file when run in a Node.js environment.
- `"mock"` : The fixed value `"index.js"` .

`node.__dirname`

`boolean | "mock"`

Default: `"mock"`

Options:

- `true` : The dirname of the **input** file relative to the `context` option.
- `false` : The regular Node.js `__dirname` behavior. The dirname of the **output** file when run in a Node.js environment.
- `"mock"` : The fixed value `"/"` .

`node.Buffer`

`boolean | "mock"`

Default: `true`

`node.setImmediate`

`boolean | "mock" | "empty"`

Default: `true`

Other node core libraries

`boolean | "mock" | "empty"`

W> This option is only activated (via `NodeSourcePlugin`) when the target is unspecified, "web" or "webworker".

Polyfills for Node.js core libraries from `node-libs-browser` are used if available, when the `NodeSourcePlugin` plugin is enabled. See the list of Node.js core libraries and their polyfills.

By default, webpack will polyfill each library if there is a known polyfill or do nothing if there is not one. In the latter case, webpack will behave as if the module name was configured with the `false` value.

T> To import a built-in module, use `__non_webpack_require__` , i.e. `__non_webpack_require__('modulename')` instead of `require('modulename')` .

Example:

```
node: {
  dns: "mock",
  fs: "empty",
  path: true,
  url: false
}
```

The `stats` option lets you precisely control what bundle information gets displayed. This can be a nice middle ground if you don't want to use `quiet` or `noInfo` because you want some bundle information, but not all of it.

T> For webpack-dev-server, this property needs to be in the `devServer` object.

W> This option does not have any effect when using the Node.js API.

`stats`

`object` `string`

There are some presets available to use as a shortcut. Use them like this:

```
stats: "errors-only"
```

Preset	Alternative	Description
`"errors-only"`	*none*	Only output when errors happen
`"minimal"`	*none*	Only output when errors or new compilation happen
`"none"`	`false`	Output nothing
`"normal"`	`true`	Standard output
`"verbose"`	*none*	Output everything

For more granular control, it is possible to specify exactly what information you want. Please note that all of the options in this object are optional.

```
stats: {

  // fallback value for stats options when an option is not defined (has precedence over local webpack defaults)

  all: undefined,

  // Add asset Information
  assets: true,

  // Sort assets by a field
  // You can reverse the sort with `!field`.
  assetsSort: "field",

  // Add build date and time information
  builtAt: true,

  // Add information about cached (not built) modules
  cached: true,

  // Show cached assets (setting this to `false` only shows emitted files)
  cachedAssets: true,

  // Add children information
  children: true,

  // Add chunk information (setting this to `false` allows for a less verbose output)
  chunks: true,

  // Add built modules information to chunk information
  chunkModules: true,

  // Add the origins of chunks and chunk merging info
  chunkOrigins: true,
```

```
// Sort the chunks by a field
// You can reverse the sort with `!field`. Default is `id`.
chunksSort: "field",

// Context directory for request shortening
context: "../src/",

// `webpack --colors` equivalent
colors: false,

// Display the distance from the entry point for each module
depth: false,

// Display the entry points with the corresponding bundles
entrypoints: false,

// Add --env information
env: false,

// Add errors
errors: true,

// Add details to errors (like resolving log)
errorDetails: true,

// Exclude assets from being displayed in stats
// This can be done with a String, a RegExp, a Function getting the assets name
// and returning a boolean or an Array of the above.
excludeAssets: "filter" | /filter/ | (assetName) => ... return true|false |
  ["filter"] | [/filter/] | [(assetName) => ... return true|false],

// Exclude modules from being displayed in stats
// This can be done with a String, a RegExp, a Function getting the modules source
// and returning a boolean or an Array of the above.
excludeModules: "filter" | /filter/ | (moduleSource) => ... return true|false |
  ["filter"] | [/filter/] | [(moduleSource) => ... return true|false],

// See excludeModules
exclude: "filter" | /filter/ | (moduleSource) => ... return true|false |
  ["filter"] | [/filter/] | [(moduleSource) => ... return true|false],

// Add the hash of the compilation
hash: true,

// Set the maximum number of modules to be shown
maxModules: 15,

// Add built modules information
modules: true,

// Sort the modules by a field
// You can reverse the sort with `!field`. Default is `id`.
modulesSort: "field",

// Show dependencies and origin of warnings/errors (since webpack 2.5.0)
moduleTrace: true,

// Show performance hint when file size exceeds `performance.maxAssetSize`
performance: true,

// Show the exports of the modules
providedExports: false,

// Add public path information
publicPath: true,

// Add information about the reasons why modules are included
reasons: true,
```

```
  // Add the source code of modules
  source: true,

  // Add timing information
  timings: true,

  // Show which exports of a module are used
  usedExports: false,

  // Add webpack version information
  version: true,

  // Add warnings
  warnings: true,

  // Filter warnings to be shown (since webpack 2.4.0),
  // can be a String, Regexp, a function getting the warning and returning a boolean
  // or an Array of a combination of the above. First match wins.
  warningsFilter: "filter" | /filter/ | ["filter", /filter/] | (warning) => ... return true|false
};
```

These are the remaining configuration options supported by webpack.

W> Help Wanted: This page is still a work in progress. If you are familiar with any of the options for which the description or examples are incomplete, please create an issue and submit a PR at the docs repo!

`amd`

`object`

Set the value of `require.amd` or `define.amd`:

```
amd: {
  jQuery: true
}
```

Certain popular modules written for AMD, most notably jQuery versions 1.7.0 to 1.9.1, will only register as an AMD module if the loader indicates it has taken special allowances for multiple versions being included on a page.

The allowances were the ability to restrict registrations to a specific version or to support different sandboxes with different defined modules.

This option allows you to set the key your module looks for to a truthy value. As it happens, the AMD support in webpack ignores the defined name anyways.

`bail`

`boolean`

Fail out on the first error instead of tolerating it. By default webpack will log these errors in red in the terminal, as well as the browser console when using HMR, but continue bundling. To enable it:

```
bail: true
```

This will force webpack to exit its bundling process.

`cache`

`boolean` `object`

Cache the generated webpack modules and chunks to improve build speed. Caching is enabled by default while in watch mode. To disable caching simply pass:

```
cache: false
```

If an object is passed, webpack will use this object for caching. Keeping a reference to this object will allow one to share the same cache between compiler calls:

```
let SharedCache = {};

export default {
  ...,
  cache: SharedCache
}
```

W> Don't share the cache between calls with different options.

?> Elaborate on the warning and example - calls with different configuration options?

`loader`

`object`

Expose custom values into the loader context.

?> Add an example...

`parallelism`

`number`

Limit the number of parallel processed modules. Can be used to fine tune performance or to get more reliable profiling results.

`profile`

`boolean`

Capture a "profile" of the application, including statistics and hints, which can then be dissected using the Analyze tool.

T> Use the StatsPlugin for more control over the generated profile.

T> Combine with `parallelism: 1` for better results.

`recordsPath`

Use this option to generate a JSON file containing webpack "records" -- pieces of data used to store module identifiers across multiple builds. You can use this file to track how modules change between builds. To generate one, simply specify a location:

```
recordsPath: path.join(__dirname, 'records.json')
```

Records are particularly useful if you have a complex setup that leverages Code Splitting. The data can be used to ensure the split bundles are achieving the caching behavior you need.

T> Note that although this file is generated by the compiler, you may still want to track it in source control to keep a history of how it has changed over time.

W> Setting `recordsPath` will essentially set `recordsInputPath` and `recordsOutputPath` to the same location. This is usually all that's necessary unless you decide to change the name of the file containing the records. See below for an example.

`recordsInputPath`

Specify the file from which to read the last set of records. This can be used to rename a records file. See the example below.

`recordsOutputPath`

Specify where the records should be written. The following example shows how you might use this option in combination with `recordsInputPath` to rename a records file:

```
recordsInputPath: path.join(__dirname, 'records.json'),
recordsOutputPath: path.join(__dirname, 'newRecords.json')
```

A variety of interfaces are available to customize the compilation process. Some features overlap between interfaces, e.g. a configuration option may be available via a CLI flag, while others exist only through a single interface. The following high-level information should get you started.

CLI

The Command Line Interface (CLI) to configure and interact with your build. It is especially useful in the case of early prototyping and profiling. For the most part, the CLI is simply used to kick off the process using a configuration file and a few flags (e.g. `--env`).

Learn more about the CLI!

Module

When processing modules with webpack, it is important to understand the different module syntaxes -- specifically the methods and variables -- that are supported.

Learn more about modules!

Node

While most users can get away with just using the CLI along with a configuration file, more fine-grained control of the compilation can be achieved via the Node interface. This includes passing multiple configurations, programmatically running or watching, and collecting stats.

Learn more about the Node API!

Loaders

Loaders are transformations that are applied to the source code of a module. They are written as functions that accept source code as a parameter and return a new version of that code with transformations applied.

Learn more about loaders!

Plugins

The plugin interface allows users to tap directly into the compilation process. Plugins can register handlers on lifecycle hooks that run at different points throughout a compilation. When each hook is executed, the plugin will have full access to the current state of the compilation.

Learn more about plugins!

For proper usage and easy distribution of this configuration, webpack can be configured with `webpack.config.js`. Any parameters sent to the CLI will map to a corresponding parameter in the config file.

Have a look at the installation guide if you don't already have webpack installed.

T> The new CLI for webpack is under development. New features are being added such as the `--init` flag. Check it out!

Usage with config file

```
webpack [--config webpack.config.js]
```

See configuration for the options in the configuration file.

Usage without config file

```
webpack <entry> [<entry>] -o <output>
```

`<entry>`

A filename or a set of named filenames which act as the entry point to build your project. You can pass multiple entries (every entry is loaded on startup). If you pass a pair in the form `<name>=<request>` you can create an additional entry point. It will be mapped to the configuration option `entry`.

`<output>`

A path and filename for the bundled file to be saved in. It will be mapped to the configuration options `output.path` and `output.filename`.

Example

If your project structure is as follows -

```
.
├── dist
├── index.html
└── src
    ├── index.js
    ├── index2.js
    └── others.js
```

```
webpack src/index.js dist/bundle.js
```

This will bundle your source code with entry as `index.js` and the output bundle file will have a path of `dist` and the filename will be `bundle.js`

```
    | Asset     | Size    | Chunks      | Chunk Names |
    |-----------|---------|-------------|-------------|
    | bundle.js | 1.54 kB | 0 [emitted] | index       |
    [0] ./src/index.js 51 bytes {0} [built]
    [1] ./src/others.js 29 bytes {0} [built]
```

```
webpack index=./src/index.js entry2=./src/index2.js dist/bundle.js
```

This will form the bundle with both the files as separate entry points.

```
    | Asset     | Size    | Chunks        | Chunk Names   |
    |-----------|---------|---------------|---------------|
    | bundle.js | 1.55 kB | 0,1 [emitted] | index, entry2 |
    [0] ./src/index.js 51 bytes {0} [built]
    [0] ./src/index2.js 54 bytes {1} [built]
    [1] ./src/others.js 29 bytes {0} {1} [built]
```

Common Options

List all of the options available on the cli

```
webpack --help
webpack -h
```

Build source using a config file

Specifies a different configuration file to pick up. Use this if you want to specify something different than `webpack.config.js`, which is the default.

```
webpack --config example.config.js
```

Print result of webpack as a JSON

```
webpack --json
webpack --json > stats.json
```

In every other case, webpack prints out a set of stats showing bundle, chunk and timing details. Using this option the output can be a JSON object. This response is accepted by webpack's analyse tool, or chrisbateman's webpack-visualizer, or th0r's webpack-bundle-analyzer. The analyse tool will take in the JSON and provide all the details of the build in graphical form.

Environment Options

When the webpack configuration exports a function, an "environment" may be passed to it.

```
webpack --env.production    # sets env.production == true
webpack --env.platform=web  # sets env.platform == "web"
```

The `--env` argument accepts various syntaxes:

Invocation	Resulting environment
`webpack --env prod`	`"prod"`
`webpack --env.prod`	`{ prod: true }`
`webpack --env.prod=1`	`{ prod: 1 }`
`webpack --env.prod=foo`	`{ prod: "foo" }`
`webpack --env.prod --env.min`	`{ prod: true, min: true }`
`webpack --env.prod --env min`	`[{ prod: true }, "min"]`
`webpack --env.prod=foo --env.prod=bar`	`{prod: [ "foo", "bar" ]}`

T> See the environment variables guide for more information on its usage.

Config Options

Parameter	Explanation	Input type	Default
`--config`	Path to the config file	string	webpack.config.js or webpackfile.js
`--config-register, -r`	Preload one or more modules before loading the webpack configuration	array	
`--config-name`	Name of the config to use	string	
`--env`	Environment passed to the config, when it is a function		
`--mode`	Mode to use, either "development" or "production"	string	

Output Options

This set of options allows you to manipulate certain output parameters of your build.

Parameter	Explanation	Input type	Default
`--output-chunk-filename`	The output filename for additional chunks	string	filename with [id] instead of [name] or [id] prefixed
`--output-filename`	The output filename of the bundle	string	[name].js
`--output-jsonp-function`	The name of the JSONP function used for chunk loading	string	webpackJsonp
`--output-library`	Expose the exports of the entry point as library	string	
`--output-library-target`	The type for exposing the exports of the entry point as library	string	var
`--output-path`	The output path for compilation assets	string	Current directory
`--output-pathinfo`	Include a comment with the request for every dependency	boolean	false
`--output-public-path`	The public path for the assets	string	/
`--output-source-map-filename`	The output filename for the SourceMap	string	[name].map or [outputFilename].map

Example Usage

```
webpack index=./src/index.js index2=./src/index2.js --output-path='./dist' --output-filename='[name][hash].bund
le.js'

| Asset                                 | Size    | Chunks      | Chunk Names   |
|---------------------------------------|---------|-------------|---------------|
| index2740fdca26e9348bedbec.bundle.js |  2.6 kB | 0 [emitted] | index2        |
| index740fdca26e9348bedbec.bundle.js  | 2.59 kB | 1 [emitted] | index         |
    [0] ./src/others.js 29 bytes {0} {1} [built]
    [1] ./src/index.js 51 bytes {1} [built]
    [2] ./src/index2.js 54 bytes {0} [built]
```

```
webpack.js index=./src/index.js index2=./src/index2.js --output-path='./dist' --output-filename='[name][hash].b
```

```
undle.js' --devtool source-map --output-source-map-filename='[name]123.map'

| Asset                                 | Size    | Chunks      | Chunk Names   |
|---------------------------------------|---------|-------------|---------------|
| index2740fdca26e9348bedbec.bundle.js | 2.76 kB | 0 [emitted] | index2        |
|  index740fdca26e9348bedbec.bundle.js | 2.74 kB | 1 [emitted] | index         |
|                       index2123.map | 2.95 kB | 0 [emitted] | index2        |
|                        index123.map | 2.95 kB | 1 [emitted] | index         |
    [0] ./src/others.js 29 bytes {0} {1} [built]
    [1] ./src/index.js 51 bytes {1} [built]
    [2] ./src/index2.js 54 bytes {0} [built]
```

Debug Options

This set of options allows you to better debug the application containing assets compiled with webpack

Parameter	Explanation	Input type	Default value
`--debug`	Switch loaders to debug mode	boolean	false
`--devtool`	Define source map type for the bundled resources	string	-
`--progress`	Print compilation progress in percentage	boolean	false
`--display-error-details`	Display details about errors	boolean	false

Module Options

These options allow you to bind modules as allowed by webpack

Parameter	Explanation	Usage
`--module-bind`	Bind an extension to a loader	`--module-bind js=babel-loader`
`--module-bind-post`	Bind an extension to a post loader	
`--module-bind-pre`	Bind an extension to a pre loader	

Watch Options

These options makes the build watch for changes in files of the dependency graph and perform the build again.

Parameter	Explanation
`--watch` , `-w`	Watch the filesystem for changes
`--watch-aggregate-timeout`	Timeout for gathering changes while watching
`--watch-poll`	The polling interval for watching (also enable polling)
`--watch-stdin` , `--stdin`	Exit the process when stdin is closed

Optimize Options

These options allow you to manipulate optimisations for a production build using webpack

Parameter	Explanation	Plugin Used
`--optimize-max-chunks`	Try to keep the chunk count below a limit	LimitChunkCountPlugin

`--optimize-min-chunk-size`	Try to keep the chunk size above a limit	MinChunkSizePlugin
`--optimize-minimize`	Minimize javascript and switches loaders to minimizing	UglifyJsPlugin & LoaderOptionsPlugin

Resolve Options

These allow you to configure the webpack resolver with aliases and extensions.

Parameter	Explanation	Example
--resolve-alias	Setup a module alias for resolving	--resolve-alias jquery-plugin=jquery.plugin
--resolve-extensions	Setup extensions that should be used to resolve modules	--resolve-extensions .es6 .js .ts
--resolve-loader-alias	Minimize javascript and switches loaders to minimizing	

Stats Options

These options allow webpack to display various stats and style them differently in the console output.

Parameter	Explanation	Type
`--color` , `--colors`	Enables/Disables colors on the console [default: (supports-color)]	boolean
`--display`	Select display preset (verbose, detailed, normal, minimal, errors-only, none; since webpack 3.0.0)	string
`--display-cached`	Display also cached modules in the output	boolean
`--display-cached-assets`	Display also cached assets in the output	boolean
`--display-chunks`	Display chunks in the output	boolean
`--display-depth`	Display distance from entry point for each module	boolean
`--display-entrypoints`	Display entry points in the output	boolean
`--display-error-details`	Display details about errors	boolean
`--display-exclude`	Exclude modules in the output	boolean
`--display-max-modules`	Sets the maximum number of visible modules in output	number
`--display-modules`	Display even excluded modules in the output	boolean
`--display-optimization-bailout`	Scope hoisting fallback trigger (since webpack 3.0.0)	boolean
`--display-origins`	Display origins of chunks in the output	boolean
`--display-provided-exports`	Display information about exports provided from modules	boolean
`--display-reasons`	Display reasons about module inclusion in the output	boolean
`--display-used-exports`	Display information about used exports in modules (Tree Shaking)	boolean
`--hide-modules`	Hides info about modules	boolean
`--sort-assets-by`	Sorts the assets list by property in asset	string
`--sort-chunks-by`	Sorts the chunks list by property in chunk	string

`--sort-modules-by`	Sorts the modules list by property in module	string
`--verbose`	Show more details	boolean

Advanced Options

Parameter	Explanation	Usage
`--bail`	Abort the compilation on first error	
`--cache`	Enable in memory caching [Enabled by default for watch]	`--cache=false`
`--define`	Define any free variable, see shimming	`--define process.env.NODE_ENV='development'`
`--hot`	Enables Hot Module Replacement	`--hot=true`
`--labeled-modules`	Enables labeled modules [Uses LabeledModulesPlugin]	
`--plugin`	Load this plugin	
`--prefetch`	Prefetch the particular file	`--prefetch=./files.js`
`--provide`	Provide these modules as globals, see shimming	`--provide jQuery=jquery`
`--records-input-path`	Path to the records file (reading)	
`--records-output-path`	Path to the records file (writing)	
`--records-path`	Path to the records file	
`--target`	The targeted execution environment	`--target='node'`

Shortcuts

Shortcut	Replaces
-d	`--debug --devtool cheap-module-eval-source-map --output-pathinfo`
-p	`--optimize-minimize --define process.env.NODE_ENV="production"` , see building for production

Profiling

The `--profile` option captures timing information for each step of the compilation and includes this in the output.

```
webpack --profile

⋮
[0] ./src/index.js 90 bytes {0} [built]
    factory:22ms building:16ms = 38ms
```

For each module, the following details are included in the output as applicable:

- `factory` : time to collect module metadata (e.g. resolving the filename)
- `building` : time to build the module (e.g. loaders and parsing)
- `dependencies` : time to identify and connect the module's dependencies

Paired with `--progress`, `--profile` gives you an in depth idea of which step in the compilation is taking how long. This can help you optimise your build in a more informed manner.

```
webpack --progress --profile

30ms building modules
1ms sealing
1ms optimizing
0ms basic module optimization
1ms module optimization
1ms advanced module optimization
0ms basic chunk optimization
0ms chunk optimization
1ms advanced chunk optimization
0ms module and chunk tree optimization
1ms module reviving
0ms module order optimization
1ms module id optimization
1ms chunk reviving
0ms chunk order optimization
1ms chunk id optimization
10ms hashing
0ms module assets processing
13ms chunk assets processing
1ms additional chunk assets processing
0ms recording
0ms additional asset processing
26ms chunk asset optimization
1ms asset optimization
6ms emitting
⋮
```

When compiling source code with webpack, users can generate a JSON file containing statistics about modules. These statistics can be used to analyze an application's dependency graph as well as to optimize compilation speed. The file is typically generated with the following CLI command:

```
webpack --profile --json > compilation-stats.json
```

The `--json > compilation-stats.json` flag indicates to webpack that it should emit the `compilation-stats.json` containing the dependency graph and various other build information. Typically, the `--profile` flag is also added so that a `profile` section is added to each `modules` object containing module-specific compilation stats.

Structure

The top-level structure of the output JSON file is fairly straightforward but there are a few nested data structures as well. Each nested structure has a dedicated section below to make this document more consumable. Note that you can click links within the top-level structure below to jump to relevant sections and documentation:

```
{
  "version": "1.4.13", // Version of webpack used for the compilation
  "hash": "11593e3b3ac85436984a", // Compilation specific hash
  "time": 2469, // Compilation time in milliseconds
  "filteredModules": 0, // A count of excluded modules when [`exclude`](/configuration/stats/#stats) is passed
to the [`toJson`](/api/node/#stats-tojson-options-) method
  "outputPath": "/", // path to webpack output directory
  "assetsByChunkName": {
    // Chunk name to emitted asset(s) mapping
    "main": "web.js?h=11593e3b3ac85436984a",
    "named-chunk": "named-chunk.web.js",
    "other-chunk": [
      "other-chunk.js",
      "other-chunk.css"
    ]
  },
  "assets": [
    // A list of [asset objects](#asset-objects)
  ],
  "chunks": [
    // A list of [chunk objects](#chunk-objects)
  ],
  "modules": [
    // A list of [module objects](#module-objects)
  ],
  "errors": [
    // A list of [error strings](#errors-and-warnings)
  ],
  "warnings": [
    // A list of [warning strings](#errors-and-warnings)
  ]
}
```

Asset Objects

Each `assets` object represents an `output` file emitted from the compilation. They all follow a similar structure:

```
{
  "chunkNames": [], // The chunks this asset contains
  "chunks": [ 10, 6 ], // The chunk IDs this asset contains
  "emitted": true, // Indicates whether or not the asset made it to the `output` directory
  "name": "10.web.js", // The `output` filename
  "size": 1058 // The size of the file in bytes
}
```

Chunk Objects

Each `chunks` object represents a group of modules known as a chunk. Each object follows the following structure:

```
{
  "entry": true, // Indicates whether or not the chunk contains the webpack runtime
  "files": [
    // An array of filename strings that contain this chunk
  ],
  "filteredModules": 0, // See the description in the [top-level structure](#structure) above
  "id": 0, // The ID of this chunk
  "initial": true, // Indicates whether this chunk is loaded on initial page load or [on demand](/guides/lazy-l
oading)
  "modules": [
    // A list of [module objects](#module-objects)
    "web.js?h=11593e3b3ac85436984a"
  ],
  "names": [
    // An list of chunk names contained within this chunk
  ],
  "origins": [
    // See the description below...
  ],
  "parents": [], // Parent chunk IDs
  "rendered": true, // Indicates whether or not the chunk went through Code Generation
  "size": 188057 // Chunk size in bytes
}
```

The `chunks` object will also contain a list of `origins` describing how the given chunk originated. Each `origins` object follows the following schema:

```
{
  "loc": "", // Lines of code that generated this chunk
  "module": "(webpack)\\test\\browsertest\\lib\\index.web.js", // Path to the module
  "moduleId": 0, // The ID of the module
  "moduleIdentifier": "(webpack)\\test\\browsertest\\lib\\index.web.js", // Path to the module
  "moduleName": "./lib/index.web.js", // Relative path to the module
  "name": "main", // The name of the chunk
  "reasons": [
    // A list of the same `reasons` found in [module objects](#module-objects)
  ]
}
```

Module Objects

What good would these statistics be without some description of the compiled application's actual modules? Each module in the dependency graph is represented by the following structure:

```
{
  "assets": [
    // A list of [asset objects](#asset-objects)
  ],
  "built": true, // Indicates that the module went through [Loaders](/concepts/loaders), Parsing, and Code Gene
ration
  "cacheable": true, // Whether or not this module is cacheable
  "chunks": [
    // IDs of chunks that contain this module
  ],
  "errors": 0, // Number of errors when resolving or processing the module
  "failed": false, // Whether or not compilation failed on this module
  "id": 0, // The ID of the module (analagous to [`module.id`](/api/module-variables#module-id-commonjs-))
```

```
  "identifier": "(webpack)\\test\\browsertest\\lib\\index.web.js", // A unique ID used internally
  "name": "./lib/index.web.js", // Path to the actual file
  "optional": false, // All requests to this module are with `try... catch` blocks (irrelevant with ESM)
  "prefetched": false, // Indicates whether or not the module was [prefetched](/plugins/prefetch-plugin)
  "profile": {
    // Module specific compilation stats corresponding to the [`--profile` flag](/api/cli#profiling) (in millis
econds)
    "building": 73, // Loading and parsing
    "dependencies": 242, // Building dependencies
    "factory": 11 // Resolving dependencies
  },
  "reasons": [
    // See the description below...
  ],
  "size": 3593, // Estimated size of the module in bytes
  "source": "// Should not break it...\r\nif(typeof...", // The stringified raw source
  "warnings": 0 // Number of warnings when resolving or processing the module
}
```

Every module also contains a list of `reasons` objects describing why that module was included in the dependency graph. Each "reason" is similar to the `origins` seen above in the chunk objects section:

```
{
  "loc": "33:24-93", // Lines of code that caused the module to be included
  "module": "./lib/index.web.js", // Relative path to the module based on [context](/configuration/entry-contex
t/#context)
  "moduleId": 0, // The ID of the module
  "moduleIdentifier": "(webpack)\\test\\browsertest\\lib\\index.web.js", // Path to the module
  "moduleName": "./lib/index.web.js", // A more readable name for the module (used for "pretty-printing")
  "type": "require.context", // The [type of request](/api/module-methods) used
  "userRequest": "../../cases" // Raw string used for the `import` or `require` request
}
```

Errors and Warnings

The `errors` and `warnings` properties each contain a list of strings. Each string contains a message and stack trace:

```
../cases/parsing/browserify/index.js
Critical dependencies:
2:114-121 This seem to be a pre-built javascript file. Even while this is possible, it's not recommended. Try t
o require to orginal source to get better results.
 @ ../cases/parsing/browserify/index.js 2:114-121
```

W> Note that the stack traces are removed when `errorDetails: false` is passed to the `toJson` method. The `errorDetails` option is set to `true` by default.

webpack provides a Node.js API which can be used directly in Node.js runtime.

The Node.js API is useful in scenarios in which you need to customize the build or development process since all the reporting and error handling must be done manually and webpack only does the compiling part. For this reason the `stats` configuration options will not have any effect in the `webpack()` call.

Installation

To start using webpack Node.js API, first install webpack if you haven't yet:

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

Then require the webpack module in your Node.js script:

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

// Or if you prefer ES2015:
import webpack from "webpack";
```

`webpack()`

The imported `webpack` function is fed a webpack Configuration Object and runs the webpack compiler if a callback function is provided:

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

webpack({
  // [Configuration Object](/configuration/)
}, (err, [stats](#stats-object)) => {
  if (err || stats.hasErrors()) {
    // [Handle errors here](#error-handling)
  }
  // Done processing
});
```

T> The `err` object **will not** include compilation errors and those must be handled separately using `stats.hasErrors()` which will be covered in detail in Error Handling section of this guide. The `err` object will only contain webpack-related issues, such as misconfiguration, etc.

T> You can provide the `webpack` function with an array of configurations. See the MultiCompiler section below for more information.

Compiler Instance

If you don't pass the `webpack` runner function a callback, it will return a webpack `Compiler` instance. This instance can be used to manually trigger the webpack runner or have it build and watch for changes, much like the CLI. The `Compiler` instance provides the following methods:

- `.run(callback)`
- `.watch(watchOptions, handler)`

Typically, only one master `Compiler` instance is created, although child compilers can be created in order to delegate specific tasks. The `Compiler` is ultimately just a function which performs bare minimum functionality to keep a lifecycle running. It delegates all the loading, bundling, and writing work to registered plugins.

The `hooks` property on a `Compiler` instance is used to register a plugin to any hook event in the `Compiler`'s lifecycle. The [`WebpackOptionsDefaulter`](https://github.com/webpack/webpack/blob/master/lib/WebpackOptionsDefaulter.js) and `WebpackOptionsApply` utilities are used by webpack to configure its `Compiler` instance with all the built-in plugins.

The `run` method is then used to kickstart all compilation work. Upon completion, the given `callback` function is executed. The final logging of stats and errors should be done in this `callback` function.

W> The API only supports a single concurrent compilation at a time. When using `run`, wait for it to finish before calling `run` or `watch` again. When using `watch`, call `close` and wait for it to finish before calling `run` or `watch` again. Concurrent compilations will corrupt the output files.

Run

Calling the `run` method on the `Compiler` instance is much like the quick run method mentioned above:

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

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

compiler.run((err, [stats](#stats-object)) => {
  // ...
});
```

Watching

Calling the `watch` method, triggers the webpack runner, but then watches for changes (much like CLI: `webpack --watch`), as soon as webpack detects a change, runs again. Returns an instance of `Watching`.

```
watch(watchOptions, callback)
```

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

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

const watching = compiler.watch({
  // Example [watchOptions](/configuration/watch/#watchoptions)
  aggregateTimeout: 300,
  poll: undefined
}, (err, [stats](#stats-object)) => {
  // Print watch/build result here...
  console.log(stats);
});
```

`Watching` options are covered in detail here.

W> Filesystem inaccuracies may trigger multiple builds for a single change. So, in the example above, the `console.log` statement may fire multiple times for a single modification. Users should expect this behavior and may check `stats.hash` to see if the file hash has actually changed.

Close `Watching`

The `watch` method returns a `Watching` instance that exposes `.close(callback)` method. Calling this method will end watching:

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

W> It's not allowed to watch or run again before the existing watcher has been closed or invalidated.

Invalidate `Watching`

Using `watching.invalidate`, you can manually invalidate the current compiling round, without stopping the watch process:

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

Stats Object

The `stats` object that is passed as a second argument of the `webpack()` callback, is a good source of information about the code compilation process. It includes:

- Errors and Warnings (if any)
- Timings
- Module and Chunk information

The webpack CLI uses this information to display nicely formatted output in your console.

T> When using the `MultiCompiler`, a `MultiStats` instance is returned that fulfills the same interface as `stats`, i.e. the methods described below.

This `stats` object exposes the following methods:

`stats.hasErrors()`

Can be used to check if there were errors while compiling. Returns `true` or `false`.

`stats.hasWarnings()`

Can be used to check if there were warnings while compiling. Returns `true` or `false`.

`stats.toJson(options)`

Returns compilation information as a JSON object. `options` can be either a string (a preset) or an object for more granular control:

```
stats.toJson("minimal"); // [more options: "verbose", etc](/configuration/stats).
```

```
stats.toJson({
  assets: false,
  hash: true
});
```

All available options and presets are described in the stats documentation.

> Here's an [example] (https://github.com/webpack/analyse/blob/master/app/pages/upload/example.json) of this function's output.

`stats.toString(options)`

Returns a formatted string of the compilation information (similar to CLI output).

Options are the same as `stats.toJson(options)` with one addition:

```
stats.toString({
  // Add console colors
  colors: true
});
```

Here's an example of `stats.toString()` usage:

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

webpack({
  // [Configuration Object](/configuration/)
}, (err, stats) => {
  if (err) {
    console.error(err);
    return;
  }

  console.log(stats.toString({
    chunks: false,  // Makes the build much quieter
    colors: true    // Shows colors in the console
  }));
});
```

MultiCompiler

The `MultiCompiler` module allows webpack to run multiple configurations in separate compilers. If the `options` parameter in the webpack's NodeJS api is an array of options, webpack applies separate compilers and calls the `callback` method at the end of each compiler execution.

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

webpack([
  { entry: './index1.js', output: { filename: 'bundle1.js' } },
  { entry: './index2.js', output: { filename: 'bundle2.js' } }
], (err, [stats](#stats-object)) => {
  process.stdout.write(stats.toString() + "\n");
})
```

W> Multiple configurations will **not be run in parallel**. Each configuration is only processed after the previous one has finished processing. To process them in parallel, you can use a third-party solution like parallel-webpack.

Error Handling

For a good error handling, you need to account for these three types of errors:

- Fatal webpack errors (wrong configuration, etc)
- Compilation errors (missing modules, syntax errors, etc)

- Compilation warnings

Here's an example that does all that:

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

webpack({
  // [Configuration Object](/configuration/)
}, (err, stats) => {
  if (err) {
    console.error(err.stack || err);
    if (err.details) {
      console.error(err.details);
    }
    return;
  }

  const info = stats.toJson();

  if (stats.hasErrors()) {
    console.error(info.errors);
  }

  if (stats.hasWarnings()) {
    console.warn(info.warnings);
  }

  // Log result...
});
```

Custom File Systems

By default, webpack reads files and writes files to disk using a normal file system. However, it is possible to change the input or output behavior using a different kind of file system (memory, webDAV, etc). To accomplish this, one can change the `inputFileSystem` or `outputFileSystem`. For example, you can replace the default `outputFileSystem` with `memory-fs` to write files to memory instead of to disk:

```
const MemoryFS = require("memory-fs");
const webpack = require("webpack");

const fs = new MemoryFS();
const compiler = webpack({ /* options*/ });

compiler.outputFileSystem = fs;
compiler.run((err, stats) => {
  // Read the output later:
  const content = fs.readFileSync("...");
});
```

Note that this is what webpack-dev-middleware, used by webpack-dev-server and many other packages, uses to mysteriously hide your files but continue serving them up to the browser!

T> The output file system you provide needs to be compatible with Node's own `fs` interface, which requires the `mkdirp` and `join` helper methods.

If Hot Module Replacement has been enabled via the `HotModuleReplacementPlugin`, its interface will be exposed under the `module.hot` property. Typically, users will check to see if the interface is accessible, then begin working with it. As an example, here's how you might `accept` an updated module:

```
if (module.hot) {
  module.hot.accept('./library.js', function() {
    // Do something with the updated library module...
  })
}
```

The following methods are supported...

`accept`

Accept updates for the given `dependencies` and fire a `callback` to react to those updates.

```
module.hot.accept(
  dependencies, // Either a string or an array of strings
  callback // Function to fire when the dependencies are updated
)
```

`decline`

Reject updates for the given `dependencies` forcing the update to fail with a `'decline'` code.

```
module.hot.decline(
  dependencies // Either a string or an array of strings
)
```

`dispose` (or `addDisposeHandler`)

Add a handler which is executed when the current module code is replaced. This should be used to remove any persistent resource you have claimed or created. If you want to transfer state to the updated module, add it to given `data` parameter. This object will be available at `module.hot.data` after the update.

```
module.hot.dispose(data => {
  // Clean up and pass data to the updated module...
})
```

`removeDisposeHandler`

Remove the callback added via `dispose` or `addDisposeHandler`.

```
module.hot.removeDisposeHandler(callback)
```

`status`

Retrieve the current status of the hot module replacement process.

```
module.hot.status() // Will return one of the following strings...
```

Status	Description

idle	The process is waiting for a call to `check` (see below)
check	The process is checking for updates
prepare	The process is getting ready for the update (e.g. downloading the updated module)
ready	The update is prepared and available
dispose	The process is calling the `dispose` handlers on the modules that will be replaced
apply	The process is calling the `accept` handlers and re-executing self-accepted modules
abort	An update was aborted, but the system is still in it's previous state
fail	An update has thrown an exception and the system's state has been compromised

`check`

Test all loaded modules for updates and, if updates exist, `apply` them.

```
module.hot.check(autoApply).then(outdatedModules => {
  // outdated modules...
}).catch(error => {
  // catch errors
});
```

The `autoApply` parameter can either be a boolean or `options` to pass to the `apply` method when called.

`apply`

Continue the update process (as long as `module.hot.status() === 'ready'`).

```
module.hot.apply(options).then(outdatedModules => {
  // outdated modules...
}).catch(error => {
  // catch errors
});
```

The optional `options` object can include the following properties:

- `ignoreUnaccepted` (boolean): Ignore changes made to unaccepted modules.
- `ignoreDeclined` (boolean): Ignore changes made to declined modules.
- `ignoreErrored` (boolean): Ignore errors throw in accept handlers, error handlers and while reevaluating module.
- `onDeclined` (function(info)): Notifier for declined modules
- `onUnaccepted` (function(info)): Notifier for unaccepted modules
- `onAccepted` (function(info)): Notifier for accepted modules
- `onDisposed` (function(info)): Notifier for disposed modules
- `onErrored` (function(info)): Notifier for errors

The `info` parameter will be an object containing some of the following values:

```
{
  type: "self-declined" | "declined" |
        "unaccepted" | "accepted" |
        "disposed" | "accept-errored" |
        "self-accept-errored" | "self-accept-error-handler-errored",
  moduleId: 4, // The module in question.
  dependencyId: 3, // For errors: the module id owning the accept handler.
  chain: [1, 2, 3, 4], // For declined/accepted/unaccepted: the chain from where the update was propagated.
  parentId: 5, // For declined: the module id of the declining parent
```

```
  outdatedModules: [1, 2, 3, 4], // For accepted: the modules that are outdated and will be disposed
  outdatedDependencies: { // For accepted: The location of accept handlers that will handle the update
    5: [4]
  },
  error: new Error(...), // For errors: the thrown error
  originalError: new Error(...) // For self-accept-error-handler-errored:
                                // the error thrown by the module before the error handler tried to handle it.
}
```

`addStatusHandler`

Register a function to listen for changes in `status`.

```
module.hot.addStatusHandler(status => {
  // React to the current status...
})
```

`removeStatusHandler`

Remove a registered status handler.

```
module.hot.removeStatusHandler(callback)
```

A loader is just a JavaScript module that exports a function. The loader runner calls this function and passes the result of the previous loader or the resource file into it. The `this` context of the function is filled-in by webpack and the loader runner with some useful methods that allow the loader (among other things) to change its invocation style to async, or get query parameters.

The first loader is passed one argument: the content of the resource file. The compiler expects a result from the last loader. The result should be a `String` or a `Buffer` (which is converted to a string), representing the JavaScript source code of the module. An optional SourceMap result (as JSON object) may also be passed.

A single result can be returned in **sync mode**. For multiple results the `this.callback()` must be called. In **async mode** `this.async()` must be called to indicate that the loader runner should wait for an asynchronous result. It returns `this.callback()`. Then the loader must return `undefined` and call that callback.

Examples

The following sections provide some basic examples of the different types of loaders. Note that the `map` and `meta` parameters are optional, see `this.callback` below.

Synchronous Loaders

Either `return` or `this.callback` can be used to return the transformed `content` synchronously:

sync-loader.js

```
module.exports = function(content, map, meta) {
  return someSyncOperation(content);
};
```

The `this.callback` method is more flexible as it allows multiple arguments to be passed as opposed to just the `content`.

sync-loader-with-multiple-results.js

```
module.exports = function(content, map, meta) {
  this.callback(null, someSyncOperation(content), map, meta);
  return; // always return undefined when calling callback()
};
```

Asynchronous Loaders

For asynchronous loaders, `this.async` is used to retrieve the `callback` function:

async-loader.js

```
module.exports = function(content, map, meta) {
  var callback = this.async();
  someAsyncOperation(content, function(err, result) {
    if (err) return callback(err);
    callback(null, result, map, meta);
  });
};
```

async-loader-with-multiple-results.js

```
module.exports = function(content, map, meta) {
  var callback = this.async();
```

```
  someAsyncOperation(content, function(err, result, sourceMaps, meta) {
    if (err) return callback(err);
    callback(null, result, sourceMaps, meta);
  });
};
```

T> Loaders were originally designed to work in synchronous loader pipelines, like Node.js (using enhanced-require), *and* asynchronous pipelines, like in webpack. However, since expensive synchronous computations are a bad idea in a single-threaded environment like Node.js, we advise to make your loader asynchronously if possible. Synchronous loaders are ok if the amount of computation is trivial.

"Raw" Loader

By default, the resource file is converted to a UTF-8 string and passed to the loader. By setting the `raw` flag, the loader will receive the raw `Buffer`. Every loader is allowed to deliver its result as `String` or as `Buffer`. The compiler converts them between loaders.

raw-loader.js

```
module.exports = function(content) {
    assert(content instanceof Buffer);
    return someSyncOperation(content);
    // return value can be a `Buffer` too
    // This is also allowed if loader is not "raw"
};
module.exports.raw = true;
```

Pitching Loader

Loaders are **always** called from right to left. There are some instances where the loader only cares about the **metadata** behind a request and can ignore the results of the previous loader. The `pitch` method on loaders is called from **left to right** before the loaders are actually executed (from right to left). For the following `use` configuration:

```
use: [
  'a-loader',
  'b-loader',
  'c-loader'
]
```

These steps would occur:

```
|- a-loader `pitch`
  |- b-loader `pitch`
    |- c-loader `pitch`
      |- requested module is picked up as a dependency
    |- c-loader normal execution
  |- b-loader normal execution
|- a-loader normal execution
```

So why might a loader take advantage of the "pitching" phase?

First, the `data` passed to the `pitch` method is exposed in the execution phase as well under `this.data` and could be useful for capturing and sharing information from earlier in the cycle.

```
module.exports = function(content) {
    return someSyncOperation(content, this.data.value);
};
```

```
module.exports.pitch = function(remainingRequest, precedingRequest, data) {
    data.value = 42;
};
```

Second, if a loader delivers a result in the `pitch` method the process turns around and skips the remaining loaders. In our example above, if the `b-loader` s `pitch` method returned something:

```
module.exports = function(content) {
  return someSyncOperation(content);
};

module.exports.pitch = function(remainingRequest, precedingRequest, data) {
  if (someCondition()) {
    return "module.exports = require(" + JSON.stringify("-!" + remainingRequest) + ");";
  }
};
```

The steps above would be shortened to:

```
|- a-loader `pitch`
  |- b-loader `pitch` returns a module
|- a-loader normal execution
```

See the bundle-loader for a good example of how this process can be used in a more meaningful way.

The Loader Context

The loader context represents the properties that are available inside of a loader assigned to the `this` property.

Given the following example this require call is used: In `/abc/file.js`:

```
require("./loader1?xyz!loader2!./resource?rrr");
```

`this.version`

Loader API version. Currently `2`. This is useful for providing backwards compatibility. Using the version you can specify custom logic or fallbacks for breaking changes.

`this.context`

The directory of the module. Can be used as context for resolving other stuff.

In the example: `/abc` because `resource.js` is in this directory

`this.request`

The resolved request string.

In the example: `"/abc/loader1.js?xyz!/abc/node_modules/loader2/index.js!/abc/resource.js?rrr"`

`this.query`

1. If the loader was configured with an `options` object, this will point to that object.
2. If the loader has no `options`, but was invoked with a query string, this will be a string starting with `?`.

W> This property is deprecated as `options` is replacing `query`. Use the `getOptions` method from `loader-utils` to extract the given loader options.

this.callback

A function that can be called synchronously or asynchronously in order to return multiple results. The expected arguments are:

```
this.callback(
  err: Error | null,
  content: string | Buffer,
  sourceMap?: SourceMap,
  meta?: any
);
```

1. The first argument must be an `Error` or `null`
2. The second argument a `string` or a `Buffer`.
3. Optional: The third argument must be a source map that is parsable by this module.
4. Optional: The fourth option, ignored by webpack, can be anything (e.g. some meta data).

T> It can be useful to pass an abstract syntax tree (AST), like `ESTree`, as the fourth argument (`meta`) to speed up the build time if you want to share common ASTs between loaders.

In case this function is called, you should return undefined to avoid ambiguous loader results.

this.async

Tells the loader-runner that the loader intends to call back asynchronously. Returns `this.callback`.

this.data

A data object shared between the pitch and the normal phase.

this.cacheable

A function that sets the cacheable flag:

```
cacheable(flag = true: boolean)
```

By default, loader results are flagged as cacheable. Call this method passing `false` to make the loader's result not cacheable.

A cacheable loader must have a deterministic result, when inputs and dependencies haven't changed. This means the loader shouldn't have other dependencies than specified with `this.addDependency`.

this.loaders

An array of all the loaders. It is writeable in the pitch phase.

```
loaders = [{request: string, path: string, query: string, module: function}]
```

In the example:

```
[
```

```
  {
    request: "/abc/loader1.js?xyz",
    path: "/abc/loader1.js",
    query: "?xyz",
    module: [Function]
  },
  {
    request: "/abc/node_modules/loader2/index.js",
    path: "/abc/node_modules/loader2/index.js",
    query: "",
    module: [Function]
  }
]
```

`this.loaderIndex`

The index in the loaders array of the current loader.

In the example: in loader1: `0`, in loader2: `1`

`this.resource`

The resource part of the request, including query.

In the example: `"/abc/resource.js?rrr"`

`this.resourcePath`

The resource file.

In the example: `"/abc/resource.js"`

`this.resourceQuery`

The query of the resource.

In the example: `"?rrr"`

`this.target`

Target of compilation. Passed from configuration options.

Example values: `"web"`, `"node"`

`this.webpack`

This boolean is set to true when this is compiled by webpack.

T> Loaders were originally designed to also work as Babel transforms. Therefore if you write a loader that works for both, you can use this property to know if there is access to additional loaderContext and webpack features.

`this.sourceMap`

Should a source map be generated. Since generating source maps can be an expensive task, you should check if source maps are actually requested.

`this.emitWarning`

```
emitWarning(warning: Error)
```

Emit a warning.

this.emitError

```
emitError(error: Error)
```

Emit an error.

this.loadModule

```
loadModule(request: string, callback: function(err, source, sourceMap, module))
```

Resolves the given request to a module, applies all configured loaders and calls back with the generated source, the sourceMap and the module instance (usually an instance of `NormalModule`). Use this function if you need to know the source code of another module to generate the result.

this.resolve

```
resolve(context: string, request: string, callback: function(err, result: string))
```

Resolve a request like a require expression.

this.addDependency

```
addDependency(file: string)
dependency(file: string) // shortcut
```

Adds a file as dependency of the loader result in order to make them watchable. For example, `html-loader` uses this technique as it finds `src` and `src-set` attributes. Then, it sets the url's for those attributes as dependencies of the html file that is parsed.

this.addContextDependency

```
addContextDependency(directory: string)
```

Add a directory as dependency of the loader result.

this.clearDependencies

```
clearDependencies()
```

Remove all dependencies of the loader result. Even initial dependencies and these of other loaders. Consider using `pitch`.

this.emitFile

```
emitFile(name: string, content: Buffer|string, sourceMap: {...})
```

Emit a file. This is webpack-specific.

`this.fs`

Access to the `compilation`'s `inputFileSystem` property.

Deprecated context properties

W> The usage of these properties is highly discouraged since we are planning to remove them from the context. They are still listed here for documentation purposes.

`this.exec`

```
exec(code: string, filename: string)
```

Execute some code fragment like a module. See this comment for a replacement method if needed.

`this.resolveSync`

```
resolveSync(context: string, request: string) -> string
```

Resolve a request like a require expression.

`this.value`

Pass values to the next loader. If you know what your result exports if executed as module, set this value here (as a only element array).

`this.inputValue`

Passed from the last loader. If you would execute the input argument as module, consider reading this variable for a shortcut (for performance).

`this.options`

The options passed to the Compiler.

`this.debug`

A boolean flag. It is set when in debug mode.

`this.minimize`

Should the result be minimized.

`this._compilation`

Hacky access to the Compilation object of webpack.

`this._compiler`

Hacky access to the Compiler object of webpack.

`this._module`

Hacky access to the Module object being loaded.

This section covers all methods available in code compiled with webpack. When using webpack to bundle your application, you can pick from a variety of module syntax styles including ES6, CommonJS, and AMD.

W> While webpack supports multiple module syntaxes, we recommend following a single syntax for consistency and to avoid odd behaviors/bugs. Here's one example of mixing ES6 and CommonJS, however there are surely others.

ES6 (Recommended)

Version 2 of webpack supports ES6 module syntax natively, meaning you can use `import` and `export` without a tool like babel to handle this for you. Keep in mind that you will still probably need babel for other ES6+ features. The following methods are supported by webpack:

`import`

Statically `import` the `export`s of another module.

```
import MyModule from './my-module.js';
import { NamedExport } from './other-module.js';
```

W> The keyword here is **statically**. Normal `import` statement cannot be used dynamically within other logic or contain variables. See the spec for more information and `import()` below for dynamic usage.

`export`

Export anything as a `default` or named export.

```
// Named exports
export var Count = 5;
export function Multiply(a, b) {
  return a * b;
}

// Default export
export default {
  // Some data...
}
```

`import()`

`import('path/to/module') -> Promise`

Dynamically load modules. Calls to `import()` are treated as split points, meaning the requested module and it's children are split out into a separate chunk.

T> The ES2015 Loader spec defines `import()` as method to load ES2015 modules dynamically on runtime.

```
if ( module.hot ) {
  import('lodash').then(_ => {
    // Do something with lodash (a.k.a '_')...
  })
}
```

W> This feature relies on `Promise` internally. If you use `import()` with older browsers, remember to shim `Promise` using a polyfill such as es6-promise or promise-polyfill.

The spec for `import` doesn't allow control over the chunk's name or other properties as "chunks" are only a concept within webpack. Luckily webpack allows some special parameters via comments so as to not break the spec:

```
import(
  /* webpackChunkName: "my-chunk-name" */
  /* webpackMode: "lazy" */
  'module'
);
```

`webpackChunkName` : A name for the new chunk. Since webpack 2.6.0, the placeholders `[index]` and `[request]` are supported within the given string to an incremented number or the actual resolved filename respectively.

`webpackMode` : Since webpack 2.6.0, different modes for resolving dynamic imports can be specified. The following options are supported:

- `"lazy"` (default): Generates a lazy-loadable chunk for each `import()` ed module.
- `"lazy-once"` : Generates a single lazy-loadable chunk that can satisfy all calls to `import()` . The chunk will be fetched on the first call to `import()` , and subsequent calls to `import()` will use the same network response. Note that this only makes sense in the case of a partially dynamic statement, e.g. `import(`./locales/${language}.json`)` , where there are multiple module paths that could potentially be requested.
- `"eager"` : Generates no extra chunk. All modules are included in the current chunk and no additional network requests are made. A `Promise` is still returned but is already resolved. In contrast to a static import, the module isn't executed until the call to `import()` is made.
- `"weak"` : Tries to load the module if the module function has already been loaded in some other way (i. e. another chunk imported it or a script containing the module was loaded). A `Promise` is still returned but, only successfully resolves if the chunks are already on the client. If the module is not available, the `Promise` is rejected. A network request will never be performed. This is useful for universal rendering when required chunks are always manually served in initial requests (embedded within the page), but not in cases where app navigation will trigger an import not initially served.

T> Note that both options can be combined like so `/* webpackMode: "lazy-once", webpackChunkName: "all-i18n-data" */` . This is parsed as a JSON5 object without curly brackets.

W> Fully dynamic statements, such as `import(foo)` , **will fail** because webpack requires at least some file location information. This is because `foo` could potentially be any path to any file in your system or project. The `import()` must contain at least some information about where the module is located, so bundling can be limited to a specific directory or set of files.

W> Every module that could potentially be requested on an `import()` call is included. For example, `import(`./locale/${language}.json`)` will cause every `.json` file in the `./locale` directory to be bundled into the new chunk. At run time, when the variable `language` has been computed, any file like `english.json` or `german.json` will be available for consumption.

W> The use of `System.import` in webpack did not fit the proposed spec, so it was deprecated in webpack 2.1.0-beta.28 in favor of `import()` .

CommonJS

The goal of CommonJS is to specify an ecosystem for JavaScript outside the browser. The following CommonJS methods are supported by webpack:

`require`

```
require(dependency: String)
```

Synchronously retrieve the exports from another module. The compiler will ensure that the dependency is available in the output bundle.

```
var $ = require("jquery");
var myModule = require("my-module");
```

W> Using it asynchronously may not have the expected effect.

`require.resolve`

```
require.resolve(dependency: String)
```

Synchronously retrieve a module's ID. The compiler will ensure that the dependency is available in the output bundle. See `module.id` for more information.

W> Module ID is a number in webpack (in contrast to NodeJS where it is a string -- the filename).

`require.cache`

Multiple requires to the same module result in only one module execution and only one export. Therefore a cache in the runtime exists. Removing values from this cache cause new module execution and a new export.

W> This is only needed in rare cases for compatibility!

```
var d1 = require("dependency");
require("dependency") === d1
delete require.cache[require.resolve("dependency")];
require("dependency") !== d1
```

```
// in file.js
require.cache[module.id] === module
require("./file.js") === module.exports
delete require.cache[module.id];
require.cache[module.id] === undefined
require("./file.js") !== module.exports // in theory; in praxis this causes a stack overflow
require.cache[module.id] !== module
```

`require.ensure`

W> `require.ensure()` is specific to webpack and superseded by `import()`.

```
require.ensure(dependencies: String[], callback: function(require), errorCallback: function(error), chunkName:
String)
```

Split out the given `dependencies` to a separate bundle that that will be loaded asynchronously. When using CommonJS module syntax, this is the only way to dynamically load dependencies. Meaning, this code can be run within execution, only loading the `dependencies` if certain conditions are met.

W> This feature relies on `Promise` internally. If you use `require.ensure` with older browsers, remember to shim `Promise` using a polyfill such as es6-promise or promise-polyfill.

```
var a = require('normal-dep');
```

```
if ( module.hot ) {
  require.ensure(['b'], function(require) {
    var c = require('c');

    // Do something special...
  });
}
```

The following parameters are supported in the order specified above:

- `dependencies` : An array of strings declaring all modules required for the code in the `callback` to execute.
- `callback` : A function that webpack will execute once the dependencies are loaded. An implementation of the `require` function is sent as a parameter to this function. The function body can use this to further `require()` modules it needs for execution.
- `errorCallback` : A function that is executed when webpack fails to load the dependencies.
- `chunkName` : A name given to the chunk created by this particular `require.ensure()` . By passing the same `chunkName` to various `require.ensure()` calls, we can combine their code into a single chunk, resulting in only one bundle that the browser must load.

W> Although the implementation of `require` is passed as an argument to the `callback` function, using an arbitrary name e.g. `require.ensure([], function(request) { request('someModule'); })` isn't handled by webpack's static parser. Use `require` instead, e.g. `require.ensure([], function(require) { require('someModule'); })` .

AMD

Asynchronous Module Definition (AMD) is a JavaScript specification that defines an interface for writing and loading modules. The following AMD methods are supported by webpack:

`define` (with factory)

```
define([name: String], [dependencies: String[]], factoryMethod: function(...))
```

If `dependencies` are provided, `factoryMethod` will be called with the exports of each dependency (in the same order). If `dependencies` are not provided, `factoryMethod` is called with `require` , `exports` and `module` (for compatibility!). If this function returns a value, this value is exported by the module. The compiler ensures that each dependency is available.

W> Note that webpack ignores the `name` argument.

```
define(['jquery', 'my-module'], function($, myModule) {
  // Do something with $ and myModule...

  // Export a function
  return function doSomething() {
    // ...
  };
});
```

W> This CANNOT be used in an asynchronous function.

`define` (with value)

```
define(value: !Function)
```

This will simply export the provided `value`. The `value` here can be anything except a function.

```
define({
  answer: 42
});
```

W> This CANNOT be used in an async function.

`require` (amd-version)

```
require(dependencies: String[], [callback: function(...)])
```

Similar to `require.ensure`, this will split the given `dependencies` into a separate bundle that will be loaded asynchronously. The `callback` will be called with the exports of each dependency in the `dependencies` array.

W> This feature relies on `Promise` internally. If you use AMD with older browsers (e.g. Internet Explorer 11), remember to shim `Promise` using a polyfill such as es6-promise or promise-polyfill.

```
require(['b'], function(b) {
  var c = require("c");
});
```

W> There is no option to provide a chunk name.

Labeled Modules

The internal `LabeledModulesPlugin` enables you to use the following methods for exporting and requiring within your modules:

`export` label

Export the given `value`. The label can occur before a function declaration or a variable declaration. The function name or variable name is the identifier under which the value is exported.

```
export: var answer = 42;
export: function method(value) {
  // Do something...
};
```

W> Using it in an async function may not have the expected effect.

`require` label

Make all exports from the dependency available in the current scope. The `require` label can occur before a string. The dependency must export values with the `export` label. CommonJS or AMD modules cannot be consumed.

some-dependency.js

```
export: var answer = 42;
export: function method(value) {
  // Do something...
};
```

```
require: 'some-dependency';
console.log(answer);
method(...);
```

Webpack

Aside from the module syntaxes described above, webpack also allows a few custom, webpack-specific methods:

`require.context`

```
require.context(directory:String, includeSubdirs:Boolean /* optional, default true */, filter:RegExp /* optiona
l */)
```

Specify a whole group of dependencies using a path to the `directory`, an option to `includeSubdirs`, and a `filter` for more fine grained control of the modules included. These can then be easily resolved later on:

```
var context = require.context('components', true, /\.html$/);
var componentA = context.resolve('componentA');
```

`require.include`

```
require.include(dependency: String)
```

Include a `dependency` without executing it. This can be used for optimizing the position of a module in the output chunks.

```
require.include('a');
require.ensure(['a', 'b'], function(require) { /* ... */ });
require.ensure(['a', 'c'], function(require) { /* ... */ });
```

This will result in following output:

- entry chunk: `file.js` and `a`
- anonymous chunk: `b`
- anonymous chunk: `c`

Without `require.include('a')` it would be duplicated in both anonymous chunks.

`require.resolveWeak`

Similar to `require.resolve`, but this won't pull the `module` into the bundle. It's what is considered a "weak" dependency.

```
if(__webpack_modules__[require.resolveWeak('module')]) {
  // Do something when module is available...
}
if(require.cache[require.resolveWeak('module')]) {
  // Do something when module was loaded before...
}

// You can perform dynamic resolves ("context")
// just as with other require/import methods.
const page = 'Foo';
__webpack_modules__[require.resolveWeak(`./page/${page}`)]
```

T> `require.resolveWeak` is the foundation of *universal rendering* (SSR + Code Splitting), as used in packages such as react-universal-component. It allows code to render synchronously on both the server and initial page-loads on the client. It requires that chunks are manually served or somehow available. It's able to require modules without indicating they should be bundled into a chunk. It's used in conjunction with `import()` which takes over when user navigation triggers additional imports.

This section covers all **variables** available in code compiled with webpack. Modules will have access to certain data from the compilation process through `module` and other variables.

`module.loaded` (NodeJS)

This is `false` if the module is currently executing, and `true` if the sync execution has finished.

`module.hot` (webpack-specific)

Indicates whether or not Hot Module Replacement is enabled and provides an interface to the process. See the HMR API page for details.

`module.id` (CommonJS)

The ID of the current module.

```
module.id === require.resolve("./file.js")
```

`module.exports` (CommonJS)

Defines the value that will be returned when a consumer makes a `require` call to the module (defaults to a new object).

```
module.exports = function doSomething() {
  // Do something...
};
```

W> This CANNOT be used in an asynchronous function.

`exports` (CommonJS)

This variable is equal to default value of `module.exports` (i.e. an object). If `module.exports` gets overwritten, `exports` will no longer be exported.

```
exports.someValue = 42;
exports.anObject = {
    x: 123
};
exports.aFunction = function doSomething() {
    // Do something
};
```

`global` (NodeJS)

See node.js global.

`process` (NodeJS)

See node.js process.

`__dirname` (NodeJS)

Depending on the config option `node.__dirname`:

- `false` : Not defined
- `mock` : equal "/"
- `true` : node.js __dirname

If used inside a expression that is parsed by the Parser, the config option is treated as `true` .

`__filename` (NodeJS)

Depending on the config option `node.__filename` :

- `false` : Not defined
- `mock` : equal "/index.js"
- `true` : node.js __filename

If used inside a expression that is parsed by the Parser, the config option is treated as `true` .

`__resourceQuery` (webpack-specific)

The resource query of the current module. If the following `require` call were made, then the query string would be available in `file.js` .

```
require('file.js?test')
```

file.js

```
__resourceQuery === '?test'
```

`__webpack_public_path__` (webpack-specific)

Equals the config options `output.publicPath` .

`__webpack_require__` (webpack-specific)

The raw require function. This expression isn't parsed by the Parser for dependencies.

`__webpack_chunk_load__` (webpack-specific)

The internal chunk loading function. Takes two arguments:

- `chunkId` The id for the chunk to load.
- `callback(require)` A callback function called once the chunk is loaded.

`__webpack_modules__` (webpack-specific)

Access to the internal object of all modules.

`__webpack_hash__` (webpack-specific)

This variable is only available with the `HotModuleReplacementPlugin` or the `ExtendedAPIPlugin` . It provides access to the hash of the compilation.

`__non_webpack_require__` (webpack-specific)

Generates a `require` function that is not parsed by webpack. Can be used to do cool stuff with a global require function if available.

`DEBUG` (webpack-specific)

Equals the config option `debug`.

This section contains guides for understanding and mastering the wide variety of tools and features that webpack offers. The first is a simple guide that takes you through installation.

The guides get more and more advanced as you go on. Most serve as a starting point, and once completed you should feel more comfortable diving into the actual documentation.

W> The output shown from running webpack in the guides may differ slightly from the output of newer versions. This is to be expected. As long as the bundles look similar and run correctly, then there shouldn't be any issues. If you do come across an example that seems to be broken by a new version, please create an issue and we will do our best to resolve the discrepancy.

This guide goes through the various methods used to install webpack.

Pre-requisites

Before we begin, make sure you have a fresh version of Node.js installed. The current Long Term Support (LTS) release is an ideal starting point. You may run into a variety of issues with the older versions as they may be missing functionality webpack and/or its related packages require.

Local Installation

The latest webpack release is:

To install the latest release or a specific version, run one of the following commands:

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

If you're using webpack 4 or later, you'll also need to install the CLI.

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

Installing locally is what we recommend for most projects. This makes it easier to upgrade projects individually when breaking changes are introduced. Typically webpack is run via one or more npm scripts which will look for a webpack installation in your local `node_modules` directory:

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

T> To run the local installation of webpack you can access its bin version as `node_modules/.bin/webpack`.

Global Installation

The following NPM installation will make `webpack` available globally:

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

W> Note that this is **not a recommended practice**. Installing globally locks you down to a specific version of webpack and could fail in projects that use a different version.

Bleeding Edge

If you are enthusiastic about using the latest that webpack has to offer, you can install beta versions or even directly from the webpack repository using the following commands:

```
npm install webpack@beta
npm install webpack/webpack#<tagname/branchname>
```

W> Take caution when installing these bleeding edge releases! They may still contain bugs and therefore should not be used in production.

Webpack is used to compile JavaScript modules. Once installed, you can interface with webpack either from its CLI or API. If you're still new to webpack, please read through the core concepts and this comparison to learn why you might use it over the other tools that are out in the community.

Basic Setup

First let's create a directory, initialize npm, install webpack locally, and install the webpack-cli (the tool used to run webpack on the command line):

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

T> Throughout the Guides we will use `diff` blocks to show you what changes we're making to directories, files, and code.

Now we'll create the following directory structure, files and their contents:

project

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

src/index.js

```
function component() {
  var 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());
```

index.html

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

We also need to adjust our `package.json` file in order to make sure we mark our package as `private`, as well as removing the `main` entry. This is to prevent an accidental publish of your code.

T> If you want to learn more about the inner workings of `package.json`, then we recommend reading the npm documentation.

package.json

```
  {
    "name": "webpack-demo",
    "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": "^4.0.1",
      "webpack-cli": "^2.0.9"
    },
    "dependencies": {}
  }
```

In this example, there are implicit dependencies between the `<script>` tags. Our `index.js` file depends on `lodash` being included in the page before it runs. This is because `index.js` never explicitly declared a need for `lodash`; it just assumes that the global variable `_` exists.

There are problems with managing JavaScript projects this way:

- It is not immediately apparent that the script depends on an external library.
- If a dependency is missing, or included in the wrong order, the application will not function properly.
- If a dependency is included but not used, the browser will be forced to download unnecessary code.

Let's use webpack to manage these scripts instead.

Creating a Bundle

First we'll tweak our directory structure slightly, separating the "source" code (`/src`) from our "distribution" code (`/dist`). The "source" code is the code that we'll write and edit. The "distribution" code is the minimized and optimized `output` of our build process that will eventually be loaded in the browser:

project

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

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

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

T> When installing a package that will be bundled into your production bundle, you should use `npm install --save`. If you're installing a package for development purposes (e.g. a linter, testing libraries, etc.) then you should use `npm install --save-dev`. More information can be found in the npm documentation.

Now, lets import `lodash` in our script:

src/index.js

```
+ import _ from 'lodash';
+
  function component() {
    var element = document.createElement('div');

-   // Lodash, currently included via a script, is required for this line to work
+   // Lodash, now imported by this script
    element.innerHTML = _.join(['Hello', 'webpack'], ' ');

    return element;
  }

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

Now, since we'll be bundling our scripts, we have to update our `index.html` file. Let's remove the lodash `<script>`, as we now `import` it, and modify the other `<script>` tag to load the bundle, instead of the raw `/src` file:

dist/index.html

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

In this setup, `index.js` explicitly requires `lodash` to be present, and binds it as `_` (no global scope pollution). By stating what dependencies a module needs, webpack can use this information to build a dependency graph. It then uses the graph to generate an optimized bundle where scripts will be executed in the correct order.

With that said, let's run `npx webpack` with our script as the entry point and `main.js` as the output. The `npx` command, which ships with Node 8.2 or higher, runs the webpack binary (`./node_modules/.bin/webpack`) of the webpack package we installed in the beginning:

```
npx webpack

Hash: dabab1bac2b940c1462b
Version: webpack 4.0.1
Time: 3003ms
Built at: 2018-2-26 22:42:11
    Asset      Size  Chunks             Chunk Names
main.js  69.6 KiB       0  [emitted]  main
Entrypoint main = main.js
   [1] (webpack)/buildin/module.js 519 bytes {0} [built]
   [2] (webpack)/buildin/global.js 509 bytes {0} [built]
   [3] ./src/index.js 256 bytes {0} [built]
    + 1 hidden module

WARNING in configuration
The 'mode' option has not been set. Set 'mode' option to 'development' or 'production' to enable defaults for t
his environment.
```

T> Your output may vary a bit, but if the build is successful then you are good to go. Also, don't worry about the warning, we'll tackle that later.

Open `index.html` in your browser and, if everything went right, you should see the following text: 'Hello webpack'.

Modules

The `import` and `export` statements have been standardized in ES2015. Although they are not supported in most browsers yet, webpack does support them out of the box.

Behind the scenes, webpack actually "transpiles" the code so that older browsers can also run it. If you inspect `dist/bundle.js`, you might be able to see how webpack does this, it's quite ingenious! Besides `import` and `export`, webpack supports various other module syntaxes as well, see Module API for more information.

Note that webpack will not alter any code other than `import` and `export` statements. If you are using other ES2015 features, make sure to use a transpiler such as Babel or Bublé via webpack's loader system.

Using a Configuration

As of version 4, webpack doesn't require any configuration, but most projects will need a more complex setup, which is why webpack supports a configuration file. This is much more efficient than having to manually type in a lot of commands in the terminal, so let's create one to replace the CLI line options used above:

project

```
  webpack-demo
  |- package.json
+ |- webpack.config.js
  |- /dist
    |- index.html
  |- /src
    |- index.js
```

webpack.config.js

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

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

Now, let's run the build again but instead using our new configuration file:

```
npx webpack --config webpack.config.js

Hash: dabab1bac2b940c1462b
Version: webpack 4.0.1
Time: 328ms
Built at: 2018-2-26 22:47:43
    Asset      Size  Chunks             Chunk Names
bundle.js  69.6 KiB       0  [emitted]  main
Entrypoint main = bundle.js
   [1] (webpack)/buildin/module.js 519 bytes {0} [built]
   [2] (webpack)/buildin/global.js 509 bytes {0} [built]
   [3] ./src/index.js 256 bytes {0} [built]
    + 1 hidden module
```

```
WARNING in configuration
The 'mode' option has not been set. Set 'mode' option to 'development' or 'production' to enable defaults for t
his environment.
```

W> Note that when calling `webpack` via its path on windows, you must use backslashes instead, e.g. `node_modules\.bin\webpack --config webpack.config.js`.

T> If a `webpack.config.js` is present, the `webpack` command picks it up by default. We use the `--config` option here only to show that you can pass a config of any name. This will be useful for more complex configurations that need to be split into multiple files.

A configuration file allows far more flexibility than simple CLI usage. We can specify loader rules, plugins, resolve options and many other enhancements this way. See the configuration documentation to learn more.

NPM Scripts

Given it's not particularly fun to run a local copy of webpack from the CLI, we can set up a little shortcut. Let's adjust our *package.json* by adding an npm script:

package.json

```
  {
    "name": "webpack-demo",
    "version": "1.0.0",
    "description": "",
    "main": "index.js",
    "scripts": {
      "test": "echo \"Error: no test specified\" && exit 1",
+     "build": "webpack"
    },
    "keywords": [],
    "author": "",
    "license": "ISC",
    "devDependencies": {
      "webpack": "^4.0.1",
      "webpack-cli": "^2.0.9",
      "lodash": "^4.17.5"
    }
  }
```

Now the `npm run build` command can be used in place of the `npx` command we used earlier. Note that within `scripts` we can reference locally installed npm packages by name the same way we did with `npx`. This convention is the standard in most npm-based projects because it allows all contributors to use the same set of common scripts (each with flags like `--config` if necessary).

Now run the following command and see if your script alias works:

```
npm run build

Hash: dabab1bac2b940c1462b
Version: webpack 4.0.1
Time: 323ms
Built at: 2018-2-26 22:50:25
    Asset      Size  Chunks             Chunk Names
bundle.js  69.6 KiB       0  [emitted]  main
Entrypoint main = bundle.js
   [1] (webpack)/buildin/module.js 519 bytes {0} [built]
   [2] (webpack)/buildin/global.js 509 bytes {0} [built]
   [3] ./src/index.js 256 bytes {0} [built]
    + 1 hidden module
```

```
WARNING in configuration
The 'mode' option has not been set. Set 'mode' option to 'development' or 'production' to enable defaults for t
his environment.
```

T> Custom parameters can be passed to webpack by adding two dashes between the `npm run build` command and your parameters, e.g. `npm run build -- --colors`.

Conclusion

Now that you have a basic build together you should move on to the next guide `Asset Management` to learn how to manage assets like images and fonts with webpack. At this point, your project should look like this:

project

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

T> If you're using npm 5, you'll probably also see a `package-lock.json` file in your directory.

If you want to learn more about webpack's design, you can check out the basic concepts and configuration pages. Furthermore, the API section digs into the various interfaces webpack offers.

If you've been following the guides from the start, you will now have a small project that shows "Hello webpack". Now let's try to incorporate some other assets, like images, to see how they can be handled.

Prior to webpack, front-end developers would use tools like grunt and gulp to process these assets and move them from their `/src` folder into their `/dist` or `/build` directory. The same idea was used for JavaScript modules, but tools like webpack will **dynamically bundle** all dependencies (creating what's known as a dependency graph). This is great because every module now *explicitly states its dependencies* and we'll avoid bundling modules that aren't in use.

One of the coolest webpack features is that you can also *include any other type of file*, besides JavaScript, for which there is a loader. This means that the same benefits listed above for JavaScript (e.g. explicit dependencies) can be applied to everything used in building a website or web app. Let's start with CSS, as you may already be familiar with that setup.

Setup

Let's make a minor change to our project before we get started:

dist/index.html

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

Loading CSS

In order to `import` a CSS file from within a JavaScript module, you need to install and add the style-loader and css-loader to your `module` configuration:

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

webpack.config.js

```
  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'
+         ]
+       }
```

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

T> webpack uses a regular expression to determine which files it should look for and serve to a specific loader. In this case any file that ends with `.css` will be served to the `style-loader` and the `css-loader`.

This enables you to `import './style.css'` into the file that depends on that styling. Now, when that module is run, a `<style>` tag with the stringified css will be inserted into the `<head>` of your html file.

Let's try it out by adding a new `style.css` file to our project and import it in our `index.js`:

project

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

src/style.css

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

src/index.js

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

  function component() {
    var 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());
```

Now run your build command:

```
npm run build

Hash: 9a3abfc96300ef87880f
Version: webpack 2.6.1
Time: 834ms
    Asset    Size  Chunks                    Chunk Names
bundle.js  560 kB       0  [emitted]  [big]  main
   [0] ./~/lodash/lodash.js 540 kB {0} [built]
   [1] ./src/style.css 1 kB {0} [built]
   [2] ./~/css-loader!./src/style.css 191 bytes {0} [built]
   [3] ./~/css-loader/lib/css-base.js 2.26 kB {0} [built]
   [4] ./~/style-loader/lib/addStyles.js 8.7 kB {0} [built]
```

```
    [5] ./~/style-loader/lib/urls.js 3.01 kB {0} [built]
    [6] (webpack)/buildin/global.js 509 bytes {0} [built]
    [7] (webpack)/buildin/module.js 517 bytes {0} [built]
    [8] ./src/index.js 351 bytes {0} [built]
```

Open up `index.html` in your browser again and you should see that `Hello webpack` is now styled in red. To see what webpack did, inspect the page (don't view the page source, as it won't show you the result) and look at the page's head tags. It should contain our style block that we imported in `index.js`.

Note that you can, and in most cases should, split your CSS for better load times in production. On top of that, loaders exist for pretty much any flavor of CSS you can think of -- postcss, sass, and less to name a few.

Loading Images

So now we're pulling in our CSS, but what about our images like backgrounds and icons? Using the file-loader we can easily incorporate those in our system as well:

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

webpack.config.js

```
  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, when you `import MyImage from './my-image.png'`, that image will be processed and added to your `output` directory *and* the `MyImage` variable will contain the final url of that image after processing. When using the css-loader, as shown above, a similar process will occur for `url('./my-image.png')` within your CSS. The loader will recognize this is a local file, and replace the `'./my-image.png'` 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.

Let's add an image to our project and see how this works, you can use any image you like:

project

```
  webpack-demo
```

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

src/index.js

```
  import _ from 'lodash';
  import './style.css';
+ import Icon from './icon.png';

  function component() {
    var 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.
+   var myIcon = new Image();
+   myIcon.src = Icon;
+
+   element.appendChild(myIcon);

    return element;
  }

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

src/style.css

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

Let's create a new build and open up the index.html file again:

```
npm run build

Hash: 854865050ea3c1c7f237
Version: webpack 2.6.1
Time: 895ms
                              Asset     Size  Chunks                    Chunk Names
5c999da72346a995e7e2718865d019c8.png  11.3 kB          [emitted]
                         bundle.js   561 kB       0  [emitted]  [big]  main
   [0] ./src/icon.png 82 bytes {0} [built]
   [1] ./~/lodash/lodash.js 540 kB {0} [built]
   [2] ./src/style.css 1 kB {0} [built]
   [3] ./~/css-loader!./src/style.css 242 bytes {0} [built]
   [4] ./~/css-loader/lib/css-base.js 2.26 kB {0} [built]
   [5] ./~/style-loader/lib/addStyles.js 8.7 kB {0} [built]
   [6] ./~/style-loader/lib/urls.js 3.01 kB {0} [built]
   [7] (webpack)/buildin/global.js 509 bytes {0} [built]
   [8] (webpack)/buildin/module.js 517 bytes {0} [built]
   [9] ./src/index.js 503 bytes {0} [built]
```

If all went well, you should now see your icon as a repeating background, as well as an `img` element beside our `Hello webpack` text. If you inspect this element, you'll see that the actual filename has changed to something like `5c999da72346a995e7e2718865d019c8.png`. This means webpack found our file in the `src` folder and processed it!

T> A logical next step from here is minifying and optimizing your images. Check out the image-webpack-loader and url-loader for more on how you can enhance your image loading process.

Loading Fonts

So what about other assets like fonts? 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. Let's update our `webpack.config.js` to handle font files:

webpack.config.js

```
  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'
+         ]
+       }
      ]
    }
  };
```

Add some font files to your project:

project

```
  webpack-demo
  |- package.json
  |- webpack.config.js
  |- /dist
    |- bundle.js
    |- index.html
  |- /src
+   |- my-font.woff
+   |- my-font.woff2
    |- icon.png
```

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

With the loader configured and fonts in place, you can incorporate them via an `@font-face` declaration. The local `url(...)` directive will be picked up by webpack just as it was with the image:

src/style.css

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

  .hello {
    color: red;
+   font-family: 'MyFont';
    background: url('./icon.png');
  }
```

Now run a new build and let's see if webpack handled our fonts:

```
npm run build

Hash: b4aef94169088c79ed1c
Version: webpack 2.6.1
Time: 775ms
                              Asset     Size  Chunks                    Chunk Names
 5c999da72346a995e7e2718865d019c8.png  11.3 kB          [emitted]
11aebbbd407bcc3ab1e914ca0238d24d.woff   221 kB          [emitted]
                          bundle.js   561 kB       0  [emitted]  [big]  main
   [0] ./src/icon.png 82 bytes {0} [built]
   [1] ./~/lodash/lodash.js 540 kB {0} [built]
   [2] ./src/style.css 1 kB {0} [built]
   [3] ./~/css-loader!./src/style.css 420 bytes {0} [built]
   [4] ./~/css-loader/lib/css-base.js 2.26 kB {0} [built]
   [5] ./src/MyFont.woff 83 bytes {0} [built]
   [6] ./~/style-loader/lib/addStyles.js 8.7 kB {0} [built]
   [7] ./~/style-loader/lib/urls.js 3.01 kB {0} [built]
   [8] (webpack)/buildin/global.js 509 bytes {0} [built]
   [9] (webpack)/buildin/module.js 517 bytes {0} [built]
  [10] ./src/index.js 503 bytes {0} [built]
```

Open up `index.html` again and see if our `Hello webpack` text has changed to the new font. If all is well, you should see the changes.

Loading Data

Another useful asset that can be loaded is data, like JSON files, CSVs, TSVs, and XML. Support for JSON is actually built-in, similar to NodeJS, meaning `import Data from './data.json'` will work by default. To import CSVs, TSVs, and XML you could use the csv-loader and xml-loader. Let's handle loading all three:

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

webpack.config.js

```
  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'
+         ]
+       }
      ]
    }
  };
```

Add some data files to your project:

project

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

src/data.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<note>
  <to>Mary</to>
  <from>John</from>
  <heading>Reminder</heading>
  <body>Call Cindy on Tuesday</body>
</note>
```

Now you can `import` any one of those four types of data (JSON, CSV, TSV, XML) and the `Data` variable you import it to will contain parsed JSON for easy consumption:

src/index.js

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

  function component() {
    var 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.
    var myIcon = new Image();
    myIcon.src = Icon;

    element.appendChild(myIcon);

+   console.log(Data);

    return element;
  }

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

When you open `index.html` and look at your console in your developer tools, you should be able to see your imported data being logged to the console!

T> This can be especially helpful when implementing some sort of data visualization using a tool like d3. Instead of making an ajax request and parsing the data at runtime you can load it into your module during the build process so that the parsed data is ready to go as soon as the module hits the browser.

Global Assets

The coolest part of everything mentioned above, is that loading assets this way allows you to group modules and assets together in a more intuitive way. Instead of relying on a global `/assets` directory that contains everything, you can group assets with the code that uses them. For example, a structure like this can be very useful:

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

This setup makes your code a lot more portable as everything that is closely coupled now lives together. Let's say you want to use `/my-component` in another project, simply copy or move it into the `/components` directory over there. As long as you've installed any *external dependencies* and your *configuration has the same loaders* defined, you should be good to go.

However, let's say you're locked into your old ways or you have some assets that are shared between multiple components (views, templates, modules, etc.). It's still possible to store these assets in a base directory and even use aliasing to make them easier to `import`.

Wrapping up

For the next guides we won't be using all the different assets we've used in this guide, so let's do some cleanup so we're prepared for the next piece of the guides Output Management:

project

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

webpack.config.js

```
  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'
-         ]
-       }
-     ]
-   }
  };
```

src/index.js

```
  import _ from 'lodash';
- import './style.css';
- import Icon from './icon.png';
- import Data from './data.xml';
-
  function component() {
    var 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.
-   var myIcon = new Image();
-   myIcon.src = Icon;
-
-   element.appendChild(myIcon);
-
-   console.log(Data);

    return element;
  }

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

Next guide

Let's move on to Output Management

Further Reading

- Loading Fonts on SurviveJS

T> This guide extends on code examples found in the `Asset Management` guide.

So far we've manually included all our assets in our `index.html` file, but as your application grows and once you start using hashes in filenames and outputting multiple bundles, it will be difficult to keep managing your `index.html` file manually. However, a few plugins exist that will make this process much easier to manage.

Preparation

First, let's adjust our project a little bit:

project

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

Let's add some logic to our `src/print.js` file:

src/print.js

```
export default function printMe() {
  console.log('I get called from print.js!');
}
```

And use that function in our `src/index.js` file:

src/index.js

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

  function component() {
    var element = document.createElement('div');
+   var 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());
```

Let's also update our `dist/index.html` file, in preparation for webpack to split out entries:

dist/index.html

```
  <!doctype html>
  <html>
    <head>
-     <title>Asset Management</title>
```

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

Now adjust the config. We'll be adding our `src/print.js` as a new entry point (`print`) and we'll change the output as well, so that it will dynamically generate bundle names, based on the entry point names:

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')
    }
  };
```

Let's run `npm run build` and see what this generates:

```
Hash: aa305b0f3373c63c9051
Version: webpack 3.0.0
Time: 536ms
          Asset     Size  Chunks                    Chunk Names
  app.bundle.js   545 kB    0, 1  [emitted]  [big]  app
print.bundle.js  2.74 kB       1  [emitted]         print
   [0] ./src/print.js 84 bytes {0} {1} [built]
   [1] ./src/index.js 403 bytes {0} [built]
   [3] (webpack)/buildin/global.js 509 bytes {0} [built]
   [4] (webpack)/buildin/module.js 517 bytes {0} [built]
    + 1 hidden module
```

We can see that webpack generates our `print.bundle.js` and `app.bundle.js` files, which we also specified in our `index.html` file. if you open `index.html` in your browser, you can see what happens when you click the button.

But what would happen if we changed the name of one of our entry points, or even added a new one? The generated bundles would be renamed on a build, but our `index.html` file would still reference the old names. Let's fix that with the `HtmlWebpackPlugin` .

Setting up HtmlWebpackPlugin

First install the plugin and adjust the `webpack.config.js` file:

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

webpack.config.js

```
  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')
    }
  };
```

Before we do a build, you should know that the `HtmlWebpackPlugin` by default will generate its own `index.html` file, even though we already have one in the `dist/` folder. This means that it will replace our `index.html` file with a newly generated one. Let's see what happens when we do an `npm run build` :

```
Hash: 81f82697c19b5f49aebd
Version: webpack 2.6.1
Time: 854ms
           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]
   [0] ./~/lodash/lodash.js 540 kB {0} [built]
   [1] (webpack)/buildin/global.js 509 bytes {0} [built]
   [2] (webpack)/buildin/module.js 517 bytes {0} [built]
   [3] ./src/index.js 172 bytes {1} [built]
   [4] multi lodash 28 bytes {0} [built]
Child html-webpack-plugin for "index.html":
       [0] ./~/lodash/lodash.js 540 kB {0} [built]
       [1] ./~/html-webpack-plugin/lib/loader.js!./~/html-webpack-plugin/default_index.ejs 538 bytes {0} [built
]
       [2] (webpack)/buildin/global.js 509 bytes {0} [built]
       [3] (webpack)/buildin/module.js 517 bytes {0} [built]
```

If you open `index.html` in your code editor, you'll see that the `HtmlWebpackPlugin` has created an entirely new file for you and that all the bundles are automatically added.

If you want to learn more about all the features and options that the `HtmlWebpackPlugin` provides, then you should read up on it on the `HtmlWebpackPlugin` repo.

You can also take a look at `html-webpack-template` which provides a couple of extra features in addition to the default template.

Cleaning up the `/dist` folder

As you might have noticed over the past guides and code example, our `/dist` folder has become quite cluttered. Webpack will generate the files and put them in the `/dist` folder for you, but it doesn't keep track of which files are actually in use by your project.

In general it's good practice to clean the `/dist` folder before each build, so that only used files will be generated. Let's take care of that.

A popular plugin to manage this is the `clean-webpack-plugin` so let's install and configure it.

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

webpack.config.js

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

Now run an `npm run build` and inspect the `/dist` folder. If everything went well you should now only see the files generated from the build and no more old files!

The Manifest

You might be wondering how webpack and its plugins seem to "know" what files are being generated. The answer is in the manifest that webpack keeps to track how all the modules map to the output bundles. If you're interested in managing webpack's `output` in other ways, the manifest would be a good place to start.

The manifest data can be extracted into a json file for easy consumption using the `WebpackManifestPlugin`.

We won't go through a full example of how to use this plugin within your projects, but you can read up on the concept page and the caching guide to find out how this ties into long term caching.

Conclusion

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 heading over to the code splitting guide.

T> This guide extends on code examples found in the Output Management guide.

If you've been following the guides, you should have a solid understanding of some of the webpack basics. Before we continue, let's look into setting up a development environment to make our lives a little easier.

W> The tools in this guide are **only meant for development**, please **avoid** using them in production!!

Using source maps

When webpack bundles your source code, it can become difficult to track down errors and warnings to their original location. For example, if you bundle three source files (`a.js` , `b.js` , and `c.js`) into one bundle (`bundle.js`) and one of the source files contains an error, the stack trace will simply point to `bundle.js` . This isn't always helpful as you probably want to know exactly which source file the error came from.

In order to make it easier to track down errors and warnings, JavaScript offers source maps, which maps your compiled code back to your original source code. If an error originates from `b.js` , the source map will tell you exactly that.

There are a lot of different options available when it comes to source maps, be sure to check them out so you can configure them to your needs.

For this guide, let's use the `inline-source-map` option, which is good for illustrative purposes (though not for production):

webpack.config.js

```
  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'
    },
+   devtool: 'inline-source-map',
    plugins: [
      new CleanWebpackPlugin(['dist']),
      new HtmlWebpackPlugin({
        title: 'Development'
      })
    ],
    output: {
      filename: '[name].bundle.js',
      path: path.resolve(__dirname, 'dist')
    }
  };
```

Now let's make sure we have something to debug, so let's create an error in our `print.js` file:

src/print.js

```
  export default function printMe() {
-   console.log('I get called from print.js!');
+   cosnole.log('I get called from print.js!');
  }
```

Run an `npm run build` , it should compile to something like this:

```
Hash: 7bf68ca15f1f2690e2d1
```

```
Version: webpack 3.1.0
Time: 1224ms
          Asset       Size  Chunks                    Chunk Names
  app.bundle.js    1.44 MB    0, 1  [emitted]  [big]  app
print.bundle.js    6.43 kB       1  [emitted]         print
     index.html  248 bytes          [emitted]
   [0] ./src/print.js 84 bytes {0} {1} [built]
   [1] ./src/index.js 403 bytes {0} [built]
   [3] (webpack)/buildin/global.js 509 bytes {0} [built]
   [4] (webpack)/buildin/module.js 517 bytes {0} [built]
    + 1 hidden module
Child html-webpack-plugin for "index.html":
       [2] (webpack)/buildin/global.js 509 bytes {0} [built]
       [3] (webpack)/buildin/module.js 517 bytes {0} [built]
        + 2 hidden modules
```

Now open the resulting `index.html` file in your browser. Click the button and look in your console where the error is displayed. The error should say something like this:

```
Uncaught ReferenceError: cosnole is not defined
    at HTMLButtonElement.printMe (print.js:2)
```

We can see that the error also contains a reference to the file (`print.js`) and line number (2) where the error occurred. This is great, because now we know exactly where to look in order to fix the issue.

Choosing a Development Tool

W> Some text editors have a "safe write" function that might interfere with some of the following tools. Read Adjusting Your text Editor for a solution to these issues.

It quickly becomes a hassle to manually run `npm run build` every time you want to compile your code.

There are a couple of different options available in webpack that help you automatically compile your code whenever it changes:

1. webpack's Watch Mode
2. webpack-dev-server
3. webpack-dev-middleware

In most cases, you probably would want to use `webpack-dev-server` , but let's explore all of the above options.

Using Watch Mode

You can instruct webpack to "watch" all files within your dependency graph for changes. If one of these files is updated, the code will be recompiled so you don't have to run the full build manually.

Let's add an npm script that will start webpack's Watch Mode:

package.json

```
  {
    "name": "development",
    "version": "1.0.0",
    "description": "",
    "main": "webpack.config.js",
    "scripts": {
      "test": "echo \"Error: no test specified\" && exit 1",
+     "watch": "webpack --watch",
      "build": "webpack"
```

```
    },
    "keywords": [],
    "author": "",
    "license": "ISC",
    "devDependencies": {
      "clean-webpack-plugin": "^0.1.16",
      "css-loader": "^0.28.4",
      "csv-loader": "^2.1.1",
      "file-loader": "^0.11.2",
      "html-webpack-plugin": "^2.29.0",
      "style-loader": "^0.18.2",
      "webpack": "^3.0.0",
      "xml-loader": "^1.2.1"
    }
  }
```

You can now run `npm run watch` from the command line to see that webpack compiles your code, but doesn't exit to the command line. This is because the script is still watching your files.

Now, with webpack watching your files, let's remove the error we introduced earlier:

src/print.js

```
  export default function printMe() {
-   cosnole.log('I get called from print.js!');
+   console.log('I get called from print.js!');
  }
```

Now save your file and check the terminal window. You should see that webpack automatically recompiles the changed module!

The only downside is that you have to refresh your browser in order to see the changes. It would be much nicer if that would happen automatically as well, so let's try `webpack-dev-server` which will do exactly that.

Using webpack-dev-server

The `webpack-dev-server` provides you with a simple web server and the ability to use live reloading. Let's set it up:

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

Change your config file to tell the dev server where to look for files:

webpack.config.js

```
  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'
    },
    devtool: 'inline-source-map',
+   devServer: {
+     contentBase: './dist'
+   },
    plugins: [
      new CleanWebpackPlugin(['dist']),
      new HtmlWebpackPlugin({
        title: 'Development'
      })
```

```
    ],
    output: {
      filename: '[name].bundle.js',
      path: path.resolve(__dirname, 'dist')
    }
  };
```

This tells `webpack-dev-server` to serve the files from the `dist` directory on `localhost:8080`.

Let's add a script to easily run the dev server as well:

package.json

```
  {
    "name": "development",
    "version": "1.0.0",
    "description": "",
    "main": "webpack.config.js",
    "scripts": {
      "test": "echo \"Error: no test specified\" && exit 1",
      "watch": "webpack --watch",
+     "start": "webpack-dev-server --open",
      "build": "webpack"
    },
    "keywords": [],
    "author": "",
    "license": "ISC",
    "devDependencies": {
      "clean-webpack-plugin": "^0.1.16",
      "css-loader": "^0.28.4",
      "csv-loader": "^2.1.1",
      "file-loader": "^0.11.2",
      "html-webpack-plugin": "^2.29.0",
      "style-loader": "^0.18.2",
      "webpack": "^3.0.0",
      "xml-loader": "^1.2.1"
    }
  }
```

Now we can run `npm start` from the command line and we will see our browser automatically loading up our page. If you now change any of the source files and save them, the web server will automatically reload after the code has been compiled. Give it a try!

The `webpack-dev-server` comes with many configurable options. Head over to the documentation to learn more.

T> Now that your server is working, you might want to give Hot Module Replacement a try!

Using webpack-dev-middleware

`webpack-dev-middleware` is a wrapper that will emit files processed by webpack to a server. This is used in `webpack-dev-server` internally, however it's available as a separate package to allow more custom setups if desired. We'll take a look at an example that combines webpack-dev-middleware with an express server.

Let's install `express` and `webpack-dev-middleware` so we can get started:

```
npm install --save-dev express webpack-dev-middleware
```

Now we need to make some adjustments to our webpack configuration file in order to make sure the middleware will function correctly:

webpack.config.js

```
  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'
    },
    devtool: 'inline-source-map',
    plugins: [
      new CleanWebpackPlugin(['dist']),
      new HtmlWebpackPlugin({
        title: 'Output Management'
      })
    ],
    output: {
      filename: '[name].bundle.js',
      path: path.resolve(__dirname, 'dist'),
+     publicPath: '/'
    }
  };
```

The `publicPath` will be used within our server script as well in order to make sure files are served correctly on `http://localhost:3000`, the port number we'll specify later. The next step is setting up our custom `express` server:

project

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

server.js

```
const express = require('express');
const webpack = require('webpack');
const webpackDevMiddleware = require('webpack-dev-middleware');

const app = express();
const config = require('./webpack.config.js');
const compiler = webpack(config);

// Tell express to use the webpack-dev-middleware and use the webpack.config.js
// configuration file as a base.
app.use(webpackDevMiddleware(compiler, {
  publicPath: config.output.publicPath
}));

// Serve the files on port 3000.
app.listen(3000, function () {
  console.log('Example app listening on port 3000!\n');
});
```

Now add an npm script to make it a little easier to run the server:

package.json

```
  {
    "name": "development",
    "version": "1.0.0",
    "description": "",
    "main": "webpack.config.js",
    "scripts": {
      "test": "echo \"Error: no test specified\" && exit 1",
      "watch": "webpack --watch",
      "start": "webpack-dev-server --open",
+     "server": "node server.js",
      "build": "webpack"
    },
    "keywords": [],
    "author": "",
    "license": "ISC",
    "devDependencies": {
      "clean-webpack-plugin": "^0.1.16",
      "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": "^3.0.0",
      "webpack-dev-middleware": "^1.12.0",
      "xml-loader": "^1.2.1"
    }
  }
```

Now in your terminal run `npm run server`, it should give you an output similar to this:

```
Example app listening on port 3000!
webpack built 27b137af6d9d8668c373 in 1198ms
Hash: 27b137af6d9d8668c373
Version: webpack 3.0.0
Time: 1198ms
          Asset       Size  Chunks                    Chunk Names
  app.bundle.js    1.44 MB    0, 1  [emitted]  [big]  app
print.bundle.js    6.57 kB       1  [emitted]         print
     index.html  306 bytes          [emitted]
   [0] ./src/print.js 116 bytes {0} {1} [built]
   [1] ./src/index.js 403 bytes {0} [built]
   [2] ./node_modules/lodash/lodash.js 540 kB {0} [built]
   [3] (webpack)/buildin/global.js 509 bytes {0} [built]
   [4] (webpack)/buildin/module.js 517 bytes {0} [built]
Child html-webpack-plugin for "index.html":
         Asset    Size  Chunks  Chunk Names
    index.html  544 kB       0
       [0] ./node_modules/html-webpack-plugin/lib/loader.js!./node_modules/html-webpack-plugin/default_index.ej
s 538 bytes {0} [built]
       [1] ./node_modules/lodash/lodash.js 540 kB {0} [built]
       [2] (webpack)/buildin/global.js 509 bytes {0} [built]
       [3] (webpack)/buildin/module.js 517 bytes {0} [built]
webpack: Compiled successfully.
```

Now fire up your browser and go to `http://localhost:3000`, you should see your webpack app running and functioning!

T> If you would like to know more about how Hot Module Replacement works, we recommend you read the Hot Module Replacement guide.

Adjusting Your Text Editor

When using automatic compilation of your code, you could run into issues when saving your files. Some editors have a "safe write" feature that can potentially interfere with recompilation.

To disable this feature in some common editors, see the list below:

- **Sublime Text 3** - Add `atomic_save: "false"` to your user preferences.
- **IntelliJ** - use search in the preferences to find "safe write" and disable it.
- **Vim** - add `:set backupcopy=yes` to your settings.
- **WebStorm** - uncheck Use `"safe write"` in `Preferences > Appearance & Behavior > System Settings`.

Conclusion

Now that you've learned how to automatically compile your code and run a simple development server, you can check out the next guide, which will cover Hot Module Replacement.

T> This guide extends on code examples found in the Development guide.

Hot Module Replacement (or HMR) is one of the most useful features offered by webpack. It allows all kinds of modules to be updated at runtime without the need for a full refresh. This page focuses on **implementation** while the concepts page gives more details on how it works and why it's useful.

W> **HMR** is not intended for use in production, meaning it should only be used in development. See the building for production guide for more information.

Enabling HMR

This feature is great for productivity. All we need to do is update our webpack-dev-server configuration, and use webpack's built in HMR plugin. We'll also remove the entry point for `print.js` as it will now be consumed by the `index.js` module.

T> If you took the route of using `webpack-dev-middleware` instead of `webpack-dev-server`, please use the `webpack-hot-middleware` package to enable HMR on your custom server or application.

webpack.config.js

```
  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']),
      new HtmlWebpackPlugin({
        title: 'Hot Module Replacement'
      }),
+     new webpack.NamedModulesPlugin(),
+     new webpack.HotModuleReplacementPlugin()
    ],
    output: {
      filename: '[name].bundle.js',
      path: path.resolve(__dirname, 'dist')
    }
  };
```

T> You can use the CLI to modify the webpack-dev-server configuration with the following command: `webpack-dev-server --hotOnly`.

Note that we've also added the `NamedModulesPlugin` to make it easier to see which dependencies are being patched. To start, we'll get the dev server up and running by executing an `npm start` from the command line.

Now let's update the `index.js` file so that when a change inside `print.js` is detected we tell webpack to accept the updated module.

index.js

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

  function component() {
    var element = document.createElement('div');
    var 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();
+   })
+ }
```

Start changing the `console.log` statement in `print.js`, and you should see the following output in the browser console.

print.js

```
  export default function printMe() {
-   console.log('I get called from print.js!');
+   console.log('Updating print.js...')
  }
```

console

```
[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
+ main.js:4330 [HMR] Consider using the NamedModulesPlugin for module names.
```

Via the Node.js API

When using Webpack Dev Server with the Node.js API, don't put the dev server options on the webpack config object. Instead, pass them as a second parameter upon creation. For example:

```
new WebpackDevServer(compiler, options)
```

To enable HMR, you also need to modify your webpack configuration object to include the HMR entry points. The `webpack-dev-server` package includes a method called `addDevServerEntrypoints` which you can use to do this. Here's a small example of how that might look:

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');
});
```

T> If you're using `webpack-dev-middleware`, check out the `webpack-hot-middleware` package to enable HMR on your custom dev server.

Gotchas

Hot Module Replacement can be tricky. To show this, let's go back to our working example. If you go ahead and click the button on the example page, you will realize the console is printing the old `printMe` function.

This is happening because the button's `onclick` event handler is still bound to the original `printMe` function.

To make this work with HMR we need to update that binding to the new `printMe` function using `module.hot.accept`:

index.js

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

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

- 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(); // Re-render the "component" to update the click handler
+     document.body.appendChild(element);
    })
  }
```

This is just one example, but there are many others that can easily trip people up. Luckily, there are a lot of loaders out there (some of which are mentioned below) that will make hot module replacement much easier.

HMR with Stylesheets

Hot Module Replacement with CSS is actually fairly straightforward with the help of the `style-loader`. This loader uses `module.hot.accept` behind the scenes to patch `<style>` tags when CSS dependencies are updated.

First let's install both loaders with the following command:

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

Now let's update the configuration file to make use of the loader.

webpack.config.js

```
  const path = require('path');
  const HtmlWebpackPlugin = require('html-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'])
      new HtmlWebpackPlugin({
        title: 'Hot Module Replacement'
      }),
      new webpack.HotModuleReplacementPlugin()
    ],
    output: {
      filename: '[name].bundle.js',
      path: path.resolve(__dirname, 'dist')
    }
  };
```

Hot loading stylesheets is as easy as importing them into a module:

project

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

```
+   | - styles.css
```

styles.css

```
body {
  background: blue;
}
```

index.js

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

  function component() {
    var element = document.createElement('div');
    var 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);
    })
  }
```

Change the style on `body` to `background: red;` and you should immediately see the page's background color change without a full refresh.

styles.css

```
  body {
-   background: blue;
+   background: red;
  }
```

Other Code and Frameworks

There are many other loaders and examples out in the community to make HMR interact smoothly with a variety of frameworks and libraries...

- React Hot Loader: Tweak react components in real time.
- Vue Loader: This loader supports HMR for vue components out of the box.
- Elm Hot Loader: Supports HMR for the Elm programming language.
- Redux HMR: No loader or plugin necessary! A simple change to your main store file is all that's required.

- Angular HMR: No loader necessary! A simple change to your main NgModule file is all that's required to have full control over the HMR APIs.

T> If you know of any other loaders or plugins that help with or enhance Hot Module Replacement please submit a pull request to add to this list!

Tree shaking is a term commonly used in the JavaScript context for dead-code elimination. It relies on the static structure of ES2015 module syntax, i.e. `import` and `export`. The name and concept have been popularized by the ES2015 module bundler rollup.

The webpack 2 release came with built-in support for ES2015 modules (alias *harmony modules*) as well as unused module export detection. The new webpack 4 release expands on this capability with a way to provide hints to the compiler via the `"sideEffects"` `package.json` flag to denote which files in your project are "pure" and therefore safe to prune if unused.

T> The remainder of this guide will stem from Getting Started. If you haven't read through that guide already, please do so now.

Add a Utility

Let's add a new utility file to our project, `src/math.js`, that exports two functions:

project

```
webpack-demo
|- package.json
|- webpack.config.js
|- /dist
  |- bundle.js
  |- index.html
|- /src
  |- index.js
+ |- math.js
|- /node_modules
```

src/math.js

```
export function square(x) {
  return x * x;
}

export function cube(x) {
  return x * x * x;
}
```

With that in place, let's update our entry script to utilize one of these new methods and remove `lodash` for simplicity:

src/index.js

```
- import _ from 'lodash';
+ import { cube } from './math.js';

  function component() {
-   var element = document.createElement('div');
+   var 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());
```

Note that we **did not `import` the `square` method** from the `src/math.js` module. That function is what's known as "dead code", meaning an unused `export` that should be dropped. Now let's run our npm script, `npm run build`, and inspect the output bundle:

dist/bundle.js (around lines 90 - 100)

```
/* 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;
}
```

Note the `unused harmony export square` comment above. If you look at the code below it, you'll notice that `square` is not being imported, however, it is still included in the bundle. We'll fix that in the next section.

Mark the file as side-effect-free

In a 100% ESM module world, identifying side effects is straightforward. However, we aren't there just yet, so in the mean time it's necessary to provide hints to webpack's compiler on the "pureness" of your code.

The way this is accomplished is the `"sideEffects"` package.json property.

```
{
  "name": "your-project",
  "sideEffects": false
}
```

All the code noted above does not contain side effects, so we can simply mark the property as `false` to inform webpack that it can safely prune unused exports.

T> A "side effect" is defined as code that performs a special behavior when imported, other than exposing one or more exports. An example of this are polyfills, which affect the global scope and usually do not provide an export.

If your code did have some side effects though, an array can be provided instead:

```
{
  "name": "your-project",
  "sideEffects": [
    "./src/some-side-effectful-file.js"
  ]
}
```

The array accepts relative, absolute, and glob patterns to the relevant files. It uses micromatch under the hood.

T> Note that any imported file is subject to tree shaking. This means if you use something like `css-loader` in your project and import a CSS file, it needs to be added to the side effect list so it will not be unintentionally dropped in production mode:

```
{
  "name": "your-project",
  "sideEffects": [
    "./src/some-side-effectful-file.js",
    "*.css"
  ]
}
```

Finally, `"sideEffects"` can also be set from the `module.rules` config option.

Minify the Output

So we've cued 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, we'll use the `-p` (production) webpack compilation flag to enable the uglifyjs minification plugin.

T> Note that the `--optimize-minimize` flag can be used to insert the `UglifyJsPlugin` as well.

As of webpack 4, this is also easily toggled via the `"mode"` config option, set to `"production"` .

webpack.config.js

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

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

T> Note that the `--optimize-minimize` flag can be used to insert the `UglifyJSPlugin` as well.

With that squared away, we can run another `npm run build` and see if anything has changed.

Notice anything different about `dist/bundle.js` ? Clearly the whole bundle is now minified and mangled, but, if you look carefully, you won't see the `square` function included but will see a mangled version of the `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.

Conclusion

So, what we've learned is that in order to take advantage of *tree shaking*, you must...

- Use ES2015 module syntax (i.e. `import` and `export`).
- Add a "sideEffects" entry to your project's `package.json` file.
- Include a minifier that supports dead code removal (e.g. the `UglifyJSPlugin`).

You can imagine your application as a tree. The source code and libraries you actually use represent the green, living leaves of the tree. Dead code represents the brown, dead leaves of the tree that are consumed by autumn. In order to get rid of the dead leaves, you have to shake the tree, causing them to fall.

If you are interested in more ways to optimize your output, please jump to the next guide for details on building for production.

In this guide we'll dive into some of the best practices and utilities for building a production site or application.

T> This walkthrough stems from Tree Shaking and Development. Please ensure you are familiar with the concepts/setup introduced in those guides before continuing on.

Setup

The goals of *development* and *production* builds differ greatly. In *development*, we want strong source mapping and a localhost server with live reloading or hot module replacement. In *production*, our goals shift to a focus on minified bundles, lighter weight source maps, and optimized assets to improve load time. With this logical separation at hand, we typically recommend writing **separate webpack configurations** for each environment.

While we will separate the *production* and *development* specific bits out, note that we'll still maintain a "common" configuration to keep things DRY. In order to merge these configurations together, we'll use a utility called `webpack-merge`. With the "common" configuration in place, we won't have to duplicate code within the environment-specific configurations.

Let's start by installing `webpack-merge` and splitting out the bits we've already worked on in previous guides:

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

project

```
  webpack-demo
  |- package.json
- |- webpack.config.js
+ |- webpack.common.js
+ |- webpack.dev.js
+ |- webpack.prod.js
  |- /dist
  |- /src
    |- index.js
    |- math.js
  |- /node_modules
```

webpack.common.js

```
+ 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']),
+     new HtmlWebpackPlugin({
+       title: 'Production'
+     })
+   ],
+   output: {
+     filename: '[name].bundle.js',
+     path: path.resolve(__dirname, 'dist')
+   }
+ };
```

webpack.dev.js

```
+ const merge = require('webpack-merge');
+ const common = require('./webpack.common.js');
+
+ module.exports = merge(common, {
+   devtool: 'inline-source-map',
+   devServer: {
+     contentBase: './dist'
+   }
+ });
```

webpack.prod.js

```
+ const merge = require('webpack-merge');
+ const UglifyJSPlugin = require('uglifyjs-webpack-plugin');
+ const common = require('./webpack.common.js');
+
+ module.exports = merge(common, {
+   plugins: [
+     new UglifyJSPlugin()
+   ]
+ });
```

In `webpack.common.js`, we now have our `entry` and `output` setup configured and we've included any plugins that are required for both environments. In `webpack.dev.js`, we've added the recommended `devtool` for that environment (strong source mapping), as well as our simple `devServer` configuration. Finally, in `webpack.prod.js`, we included the `UglifyJSPlugin` which was first introduced by the tree shaking guide.

Note the use of `merge()` in the environment-specific configurations to easily include our common configuration in `dev` and `prod`. The `webpack-merge` tool offers a variety of advanced features for merging but for our use case we won't need any of that.

NPM Scripts

Now let's repoint our `scripts` to the new configurations. We'll use the *development* one for our `webpack-dev-server`, `npm start`, script and the *production* one for our `npm run build` script:

package.json

```
  {
    "name": "development",
    "version": "1.0.0",
    "description": "",
    "main": "webpack.config.js",
    "scripts": {
-     "start": "webpack-dev-server --open",
+     "start": "webpack-dev-server --open --config webpack.dev.js",
-     "build": "webpack"
+     "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": "^3.0.0",
      "webpack-dev-middleware": "^1.12.0",
      "webpack-dev-server": "^2.9.1",
      "webpack-merge": "^4.1.0",
      "xml-loader": "^1.2.1"
    }
  }
```

Feel free to run those scripts and see how the output changes as we continue adding to our *production* configuration.

Minification

Note that while the `UglifyJSPlugin` is a great place to start for minification, there are other options out there. Here are a few more popular ones:

- `BabelMinifyWebpackPlugin`
- `ClosureCompilerPlugin`

If you decide to try another, just make sure your new choice also drops dead code as described in the tree shaking guide.

Source Mapping

We encourage you to have source maps enabled in production, as they are useful for debugging as well as running benchmark tests. That said, you should choose one with a fairly quick build speed that's recommended for production use (see `devtool`). For this guide, we'll use the `source-map` option in *production* as opposed to the `inline-source-map` we used in *development*:

webpack.prod.js

```
  const merge = require('webpack-merge');
  const UglifyJSPlugin = require('uglifyjs-webpack-plugin');
  const common = require('./webpack.common.js');

  module.exports = merge(common, {
+   devtool: 'source-map',
    plugins: [
-     new UglifyJSPlugin()
+     new UglifyJSPlugin({
+       sourceMap: true
+     })
    ]
  });
```

T> Avoid `inline-***` and `eval-***` use in production as they can increase bundle size and reduce the overall performance.

Specify the Environment

Many libraries will key off the `process.env.NODE_ENV` variable to determine what should be included in the library. For example, when not in *production* some libraries may add additional logging and testing to make debugging easier. However, with `process.env.NODE_ENV === 'production'` they might drop or add significant portions of code to optimize how things run for your actual users. We can use webpack's built in `DefinePlugin` to define this variable for all our dependencies:

webpack.prod.js

```
+ const webpack = require('webpack');
  const merge = require('webpack-merge');
  const UglifyJSPlugin = require('uglifyjs-webpack-plugin');
  const common = require('./webpack.common.js');

  module.exports = merge(common, {
    devtool: 'source-map',
    plugins: [
      new UglifyJSPlugin({
        sourceMap: true
-     })
+     }),
+     new webpack.DefinePlugin({
+       'process.env.NODE_ENV': JSON.stringify('production')
+     })
    ]
  });
```

T> Technically, `NODE_ENV` is a system environment variable that Node.js exposes into running scripts. It is used by convention to determine dev-vs-prod behavior by server tools, build scripts, and client-side libraries. Contrary to expectations, `process.env.NODE_ENV` is not set to `"production"` **within** the build script `webpack.config.js`, see #2537. Thus, conditionals like `process.env.NODE_ENV === 'production' ? '[name].[hash].bundle.js' : '[name].bundle.js'` within webpack configurations do not work as expected.

If you're using a library like `react`, you should actually see a significant drop in bundle size after adding this plugin. Also note that any of our local `/src` code can key off of this as well, so the following check would be valid:

src/index.js

```
  import { cube } from './math.js';
+
+ if (process.env.NODE_ENV !== 'production') {
+   console.log('Looks like we are in development mode!');
+ }

  function component() {
    var element = document.createElement('pre');

    element.innerHTML = [
      'Hello webpack!',
      '5 cubed is equal to ' + cube(5)
    ].join('\n\n');

    return element;
  }

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

Split CSS

As mentioned in **Asset Management** at the end of the Loading CSS section, it is typically best practice to split your CSS out to a separate file using the `ExtractTextPlugin`. There are some good examples of how to do this in the plugin's documentation. The `disable` option can be used in combination with the `--env` flag to allow inline loading in development, which is recommended for Hot Module Replacement and build speed.

CLI Alternatives

Some of what has been described above is also achievable via the command line. For example, the `--optimize-minimize` flag will include the `UglifyJSPlugin` behind the scenes. The `--define process.env.NODE_ENV="'production'"` will do the same for the `DefinePlugin` instance described above. And, `webpack -p` will automatically invoke both those flags and thus the plugins to be included.

While these short hand methods are nice, we usually recommend just using the configuration as it's better to understand exactly what is being done for you in both cases. The configuration also gives you more control on fine tuning other options within both plugins.

T> This guide extends the examples provided in Getting Started and Output Management. Please make sure you are at least familiar with the examples provided in them.

Code splitting is one of the most compelling features of webpack. This feature allows you to split your code into various bundles which can then be loaded on demand or in parallel. It can be used to achieve smaller bundles and control resource load prioritization which, if used correctly, can have a major impact on load time.

There are three general approaches to code splitting available:

- Entry Points: Manually split code using `entry` configuration.
- Prevent Duplication: Use the `CommonsChunkPlugin` to dedupe and split chunks.
- Dynamic Imports: Split code via inline function calls within modules.

Entry Points

This is by far the easiest, and most intuitive, way to split code. However, it is more manual and has some pitfalls we will go over. Let's take a look at how we might split another module from the main bundle:

project

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

another-module.js

```
import _ from 'lodash';

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

webpack.config.js

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

module.exports = {
  entry: {
    index: './src/index.js',
    another: './src/another-module.js'
  },
  plugins: [
    new HTMLWebpackPlugin({
      title: 'Code Splitting'
    })
  ],
  output: {
    filename: '[name].bundle.js',
    path: path.resolve(__dirname, 'dist')
  }
};
```

This will yield the following build result:

```
Hash: 309402710a14167f42a8
Version: webpack 2.6.1
Time: 570ms
            Asset    Size  Chunks                    Chunk Names
  index.bundle.js  544 kB       0  [emitted]  [big]  index
another.bundle.js  544 kB       1  [emitted]  [big]  another
   [0] ./~/lodash/lodash.js 540 kB {0} {1} [built]
   [1] (webpack)/buildin/global.js 509 bytes {0} {1} [built]
   [2] (webpack)/buildin/module.js 517 bytes {0} {1} [built]
   [3] ./src/another-module.js 87 bytes {1} [built]
   [4] ./src/index.js 216 bytes {0} [built]
```

As mentioned there are some pitfalls to this approach:

- If there are any duplicated modules between entry chunks they will be included in both bundles.
- It isn't as flexible and can't be used to dynamically split code with the core application logic.

The first of these two points is definitely an issue for our example, as `lodash` is also imported within `./src/index.js` and will thus be duplicated in both bundles. Let's remove this duplication by using the `CommonsChunkPlugin`.

Prevent Duplication

The `CommonsChunkPlugin` allows us to extract common dependencies into an existing entry chunk or an entirely new chunk. Let's use this to de-duplicate the `lodash` dependency from the previous example:

webpack.config.js

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

  module.exports = {
    entry: {
      index: './src/index.js',
      another: './src/another-module.js'
    },
    plugins: [
      new HTMLWebpackPlugin({
        title: 'Code Splitting'
-     })
+     }),
+     new webpack.optimize.CommonsChunkPlugin({
+       name: 'common' // Specify the common bundle's name.
+     })
    ],
    output: {
      filename: '[name].bundle.js',
      path: path.resolve(__dirname, 'dist')
    }
  };
```

With the `CommonsChunkPlugin` in place, we should now see the duplicate dependency removed from our `index.bundle.js`. The plugin should notice that we've separated `lodash` out to a separate chunk and remove the dead weight from our main bundle. Let's do an `npm run build` to see if it worked:

```
Hash: 70a59f8d46ff12575481
Version: webpack 2.6.1
Time: 510ms
            Asset       Size  Chunks                    Chunk Names
  index.bundle.js  665 bytes       0  [emitted]         index
another.bundle.js  537 bytes       1  [emitted]         another
```

```
  common.bundle.js     547 kB       2  [emitted]  [big]  common
    [0] ./~/lodash/lodash.js 540 kB {2} [built]
    [1] (webpack)/buildin/global.js 509 bytes {2} [built]
    [2] (webpack)/buildin/module.js 517 bytes {2} [built]
    [3] ./src/another-module.js 87 bytes {1} [built]
    [4] ./src/index.js 216 bytes {0} [built]
```

Here are some other useful plugins and loaders provided by the community for splitting code:

- `ExtractTextPlugin`: Useful for splitting CSS out from the main application.
- `bundle-loader`: Used to split code and lazy load the resulting bundles.
- `promise-loader`: Similar to the `bundle-loader` but uses promises.

T> The `CommonsChunkPlugin` is also used to split vendor modules from core application code using explicit vendor chunks.

Dynamic Imports

Two similar techniques are supported by webpack when it comes to dynamic code splitting. The first and more preferable approach is to use the `import()` syntax that conforms to the ECMAScript proposal for dynamic imports. The legacy, webpack-specific approach is to use `require.ensure`. Let's try using the first of these two approaches...

W> `import()` calls use promises internally. If you use `import()` with older browsers, remember to shim `Promise` using a polyfill such as es6-promise or promise-polyfill.

Before we start, let's remove the extra `entry` and `CommonsChunkPlugin` from our config as they won't be needed for this next demonstration:

webpack.config.js

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

  module.exports = {
    entry: {
+     index: './src/index.js'
-     index: './src/index.js',
-     another: './src/another-module.js'
    },
    plugins: [
      new HTMLWebpackPlugin({
        title: 'Code Splitting'
-     }),
+     })
-     new webpack.optimize.CommonsChunkPlugin({
-       name: 'common' // Specify the common bundle's name.
-     })
    ],
    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. For more information on `chunkFilename`, see output documentation. We'll also update our project to remove the now unused files:

project

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

Now, instead of statically importing `lodash`, we'll use dynamic importing to separate a chunk:

src/index.js

```
- import _ from 'lodash';
-
- function component() {
+ function getComponent() {
-   var element = document.createElement('div');
-
-   // Lodash, now imported by this script
-   element.innerHTML = _.join(['Hello', 'webpack'], ' ');
+   return import(/* webpackChunkName: "lodash" */ 'lodash').then(_ => {
+     var element = document.createElement('div');
+
+     element.innerHTML = _.join(['Hello', 'webpack'], ' ');
+
+     return element;
+
+   }).catch(error => 'An error occurred while loading the component');
  }

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

Note the use of `webpackChunkName` in the comment. This will cause our separate bundle to be named `lodash.bundle.js` instead of just `[id].bundle.js`. For more information on `webpackChunkName` and the other available options, see the `import()` documentation. Let's run webpack to see `lodash` separated out to a separate bundle:

```
Hash: a27e5bf1dd73c675d5c9
Version: webpack 2.6.1
Time: 544ms
           Asset     Size  Chunks                    Chunk Names
lodash.bundle.js   541 kB       0  [emitted]  [big]  lodash
 index.bundle.js  6.35 kB       1  [emitted]         index
   [0] ./~/lodash/lodash.js 540 kB {0} [built]
   [1] ./src/index.js 377 bytes {1} [built]
   [2] (webpack)/buildin/global.js 509 bytes {0} [built]
   [3] (webpack)/buildin/module.js 517 bytes {0} [built]
```

As `import()` returns a promise, it can be used with `async` functions. However, this requires using a pre-processor like Babel and the Syntax Dynamic Import Babel Plugin. Here's how it would simplify the code:

src/index.js

```
- function getComponent() {
+ async function getComponent() {
-   return import(/* webpackChunkName: "lodash" */ 'lodash').then(_ => {
-     var element = document.createElement('div');
-
-     element.innerHTML = _.join(['Hello', 'webpack'], ' ');
-
```

```
-     return element;
-
-   }).catch(error => 'An error occurred while loading the component');
+   var element = document.createElement('div');
+   const _ = await import(/* webpackChunkName: "lodash" */ 'lodash');
+
+   element.innerHTML = _.join(['Hello', 'webpack'], ' ');
+
+   return element;
  }

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

Bundle Analysis

Once you start splitting your code, it can be useful to analyze the output to check where modules have ended up. The official analyze tool is a good place to start. There are some other community-supported options out there as well:

- webpack-chart: Interactive pie chart for webpack stats.
- webpack-visualizer: Visualize and analyze your bundles to see which modules are taking up space and which might be duplicates.
- webpack-bundle-analyzer: A plugin and CLI utility that represents bundle content as convenient interactive zoomable treemap.

Next Steps

See Lazy Loading for a more concrete example of how `import()` can be used in a real application and Caching to learn how to split code more effectively.

T> This guide is a small follow-up to Code Splitting. If you have not yet read through that guide, please do so now.

Lazy, or "on demand", loading is a great way to optimize your site or application. This practice essentially involves splitting your code at logical breakpoints, and then loading it once the user has done something that requires, or will require, a new block of code. This speeds up the initial load of the application and lightens its overall weight as some blocks may never even be loaded.

Example

Let's take the example from Code Splitting and tweak it a bit to demonstrate this concept even more. The code there does cause a separate chunk, `lodash.bundle.js`, to be generated and technically "lazy-loads" it as soon as the script is run. The trouble is that no user interaction is required to load the bundle -- meaning that every time the page is loaded, the request will fire. This doesn't help us too much and will impact performance negatively.

Let's try something different. We'll add an interaction to log some text to the console when the user clicks a button. However, we'll wait to load that code (`print.js`) until the interaction occurs for the first time. To do this we'll go back and rework the final *Dynamic Imports* example from *Code Splitting* and leave `lodash` in the main chunk.

project

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

src/print.js

```
console.log('The print.js module has loaded! See the network tab in dev tools...');

export default () => {
  console.log('Button Clicked: Here\'s "some text"!');
}
```

src/index.js

```
+ import _ from 'lodash';
+
- async function getComponent() {
+ function component() {
    var element = document.createElement('div');
-   const _ = await import(/* webpackChunkName: "lodash" */ 'lodash');
+   var button = document.createElement('button');
+   var br = document.createElement('br');

+   button.innerHTML = 'Click me and look at the console!';
    element.innerHTML = _.join(['Hello', 'webpack'], ' ');
+   element.appendChild(br);
+   element.appendChild(button);
+
+   // Note that because a network request is involved, some indication
+   // of loading would need to be shown in a production-level site/app.
+   button.onclick = e => import(/* webpackChunkName: "print" */ './print').then(module => {
+     var print = module.default;
+
+     print();
+   });
```

```
    return element;
  }

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

W> Note that when using `import()` on ES6 modules you must reference the `.default` property as it's the actual `module` object that will be returned when the promise is resolved.

Now let's run webpack and check out our new lazy-loading functionality:

```
Hash: e0f95cc0bda81c2a1340
Version: webpack 3.0.0
Time: 1378ms
          Asset       Size  Chunks                    Chunk Names
print.bundle.js  417 bytes       0  [emitted]         print
index.bundle.js     548 kB       1  [emitted]  [big]  index
     index.html  189 bytes          [emitted]
   [0] ./src/index.js 742 bytes {1} [built]
   [2] (webpack)/buildin/global.js 509 bytes {1} [built]
   [3] (webpack)/buildin/module.js 517 bytes {1} [built]
   [4] ./src/print.js 165 bytes {0} [built]
    + 1 hidden module
Child html-webpack-plugin for "index.html":
       [2] (webpack)/buildin/global.js 509 bytes {0} [built]
       [3] (webpack)/buildin/module.js 517 bytes {0} [built]
        + 2 hidden modules
```

Frameworks

Many frameworks and libraries have their own recommendations on how this should be accomplished within their methodologies. Here are a few examples:

- React: Code Splitting and Lazy Loading
- Vue: Lazy Load in Vue using Webpack's code splitting
- AngularJS: AngularJS + Webpack = lazyLoad by @var_bincom

T> The examples in this guide stem from getting started, output management and code splitting.

So we're using webpack to bundle our modular application which yields a deployable `/dist` directory. Once the contents of `/dist` have been deployed to a server, clients (typically browsers) will hit that server to grab the site and its assets. The last step can be time consuming, which is why browsers use a technique called caching. This allows sites to load faster with less unnecessary network traffic, however it can also cause headaches when you need new code to be picked up.

This guide focuses on the configuration needed to ensure files produced by webpack compilation can remain cached unless their contents has changed.

Output Filenames

A simple way to ensure the browser picks up changed files is by using `output.filename` substitutions. The `[hash]` substitution can be used to include a build-specific hash in the filename, however it's even better to use the `[chunkhash]` substitution which includes a chunk-specific hash in the filename.

Let's get our project set up using the example from getting started with the `plugins` from output management, so we don't have to deal with maintaining our `index.html` file manually:

project

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

webpack.config.js

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

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

Running our build script, `npm run build`, with this configuration should produce the following output:

```
Hash: f7a289a94c5e4cd1e566
Version: webpack 3.5.1
Time: 835ms
                        Asset       Size  Chunks                    Chunk Names
main.7e2c49a622975ebd9b7e.js     544 kB       0  [emitted]  [big]  main
                 index.html  197 bytes          [emitted]
```

```
    [0] ./src/index.js 216 bytes {0} [built]
    [2] (webpack)/buildin/global.js 509 bytes {0} [built]
    [3] (webpack)/buildin/module.js 517 bytes {0} [built]
     + 1 hidden module
Child html-webpack-plugin for "index.html":
     1 asset
       [2] (webpack)/buildin/global.js 509 bytes {0} [built]
       [3] (webpack)/buildin/module.js 517 bytes {0} [built]
        + 2 hidden modules
```

As you can see the bundle's name now reflects its content (via the hash). If we run another build without making any changes, we'd expect that filename to stay the same. However, if we were to run it again, we may find that this is not the case:

```
Hash: f7a289a94c5e4cd1e566
Version: webpack 3.5.1
Time: 835ms
                        Asset       Size  Chunks                    Chunk Names
main.205199ab45963f6a62ec.js     544 kB       0  [emitted]  [big]  main
                 index.html  197 bytes          [emitted]
   [0] ./src/index.js 216 bytes {0} [built]
   [2] (webpack)/buildin/global.js 509 bytes {0} [built]
   [3] (webpack)/buildin/module.js 517 bytes {0} [built]
    + 1 hidden module
Child html-webpack-plugin for "index.html":
     1 asset
       [2] (webpack)/buildin/global.js 509 bytes {0} [built]
       [3] (webpack)/buildin/module.js 517 bytes {0} [built]
        + 2 hidden modules
```

This is because webpack includes certain boilerplate, specifically the runtime and manifest, in the entry chunk.

W> Output may differ depending on your current webpack version. Newer versions may not have all the same issues with hashing as some older versions, but we still recommend the following steps to be safe.

Extracting Boilerplate

As we learned in code splitting, the `CommonsChunkPlugin` can be used to split modules out into separate bundles. A lesser-known feature of the `CommonsChunkPlugin` is extracting webpack's boilerplate and manifest which can change with every build. By specifying a name not mentioned in the `entry` configuration, the plugin will automatically extract what we want into a separate bundle:

webpack.config.js

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

  module.exports = {
    entry: './src/index.js',
    plugins: [
      new CleanWebpackPlugin(['dist']),
      new HtmlWebpackPlugin({
        title: 'Caching'
-     })
+     }),
+     new webpack.optimize.CommonsChunkPlugin({
+       name: 'manifest'
+     })
    ],
```

```
    output: {
      filename: '[name].[chunkhash].js',
      path: path.resolve(__dirname, 'dist')
    }
  };
```

Let's run another build to see the extracted `manifest` bundle:

```
Hash: 80552632979856ddab34
Version: webpack 3.3.0
Time: 1512ms
                          Asset       Size  Chunks                    Chunk Names
    main.5ec8e954e32d66dee1aa.js     542 kB       0  [emitted]  [big]  main
manifest.719796322be98041fff2.js    5.82 kB       1  [emitted]         manifest
                     index.html  275 bytes          [emitted]
   [0] ./src/index.js 336 bytes {0} [built]
   [2] (webpack)/buildin/global.js 509 bytes {0} [built]
   [3] (webpack)/buildin/module.js 517 bytes {0} [built]
    + 1 hidden module
```

It's also good practice to extract third-party libraries, such as `lodash` or `react`, to a separate `vendor` chunk as they are less likely to change than our local source code. This step will allow clients to request even less from the server to stay up to date. This can be done by using a combination of a new `entry` point along with another `CommonsChunkPlugin` instance:

webpack.config.js

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

  module.exports = {
-   entry: './src/index.js',
+   entry: {
+     main: './src/index.js',
+     vendor: [
+       'lodash'
+     ]
+   },
    plugins: [
      new CleanWebpackPlugin(['dist']),
      new HtmlWebpackPlugin({
        title: 'Caching'
      }),
+     new webpack.optimize.CommonsChunkPlugin({
+       name: 'vendor'
+     }),
      new webpack.optimize.CommonsChunkPlugin({
        name: 'manifest'
      })
    ],
    output: {
      filename: '[name].[chunkhash].js',
      path: path.resolve(__dirname, 'dist')
    }
  };
```

W> Note that order matters here. The `'vendor'` instance of the `CommonsChunkPlugin` must be included prior to the `'manifest'` instance.

Let's run another build to see our new `vendor` bundle:

```
Hash: 69eb92ebf8935413280d
Version: webpack 3.3.0
Time: 1502ms
                           Asset       Size  Chunks                    Chunk Names
  vendor.8196d409d2f988123318.js     541 kB       0  [emitted]  [big]  vendor
    main.0ac0ae2d4a11214ccd19.js  791 bytes       1  [emitted]         main
manifest.004a1114de8bcf026622.js    5.85 kB       2  [emitted]         manifest
                     index.html  352 bytes          [emitted]
   [1] ./src/index.js 336 bytes {1} [built]
   [2] (webpack)/buildin/global.js 509 bytes {0} [built]
   [3] (webpack)/buildin/module.js 517 bytes {0} [built]
   [4] multi lodash 28 bytes {0} [built]
    + 1 hidden module
```

Module Identifiers

Let's add another module, `print.js`, to our project:

project

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

print.js

```
+ export default function print(text) {
+   console.log(text);
+ };
```

src/index.js

```
  import _ from 'lodash';
+ import Print from './print';

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

    // Lodash, now imported by this script
    element.innerHTML = _.join(['Hello', 'webpack'], ' ');
+   element.onclick = Print.bind(null, 'Hello webpack!');

    return element;
  }

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

Running another build, we would expect only our `main` bundle's hash to change, however...

```
Hash: d38a06644fdbb898d795
Version: webpack 3.3.0
Time: 1445ms
                           Asset       Size  Chunks                    Chunk Names
  vendor.a7561fb0e9a071baadb9.js     541 kB       0  [emitted]  [big]  vendor
    main.b746e3eb72875af2caa9.js    1.22 kB       1  [emitted]         main
```

```
manifest.1400d5af64fc1b7b3a45.js    5.85 kB       2  [emitted]         manifest
                     index.html  352 bytes          [emitted]
   [1] ./src/index.js 421 bytes {1} [built]
   [2] (webpack)/buildin/global.js 509 bytes {0} [built]
   [3] (webpack)/buildin/module.js 517 bytes {0} [built]
   [4] ./src/print.js 62 bytes {1} [built]
   [5] multi lodash 28 bytes {0} [built]
    + 1 hidden module
```

... we can see that all three have. This is because each `module.id` is incremented based on resolving order by default. Meaning when the order of resolving is changed, the IDs will be changed as well. So, to recap:

- The `main` bundle changed because of its new content.
- The `vendor` bundle changed because its `module.id` was changed.
- And, the `manifest` bundle changed because it now contains a reference to a new module.

The first and last are expected -- it's the `vendor` hash we want to fix. Luckily, there are two plugins we can use to resolve this issue. The first is the `NamedModulesPlugin`, which will use the path to the module rather than a numerical identifier. While this plugin is useful during development for more readable output, it does take a bit longer to run. The second option is the `HashedModuleIdsPlugin`, which is recommended for production builds:

webpack.config.js

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

  module.exports = {
    entry: {
      main: './src/index.js',
      vendor: [
        'lodash'
      ]
    },
    plugins: [
      new CleanWebpackPlugin(['dist']),
      new HtmlWebpackPlugin({
        title: 'Caching'
      }),
+     new webpack.HashedModuleIdsPlugin(),
      new webpack.optimize.CommonsChunkPlugin({
        name: 'vendor'
      }),
      new webpack.optimize.CommonsChunkPlugin({
        name: 'manifest'
      })
    ],
    output: {
      filename: '[name].[chunkhash].js',
      path: path.resolve(__dirname, 'dist')
    }
  };
```

Now, despite any new local dependencies, our `vendor` hash should stay consistent between builds:

```
Hash: 1f49b42afb9a5acfbaff
Version: webpack 3.3.0
Time: 1372ms
                          Asset       Size  Chunks                    Chunk Names
  vendor.eed6dcc3b30cfa138aaa.js     541 kB       0  [emitted]  [big]  vendor
    main.d103ac311788fcb7e329.js    1.22 kB       1  [emitted]         main
manifest.d2a6dc1ccece13f5a164.js    5.85 kB       2  [emitted]         manifest
                     index.html  352 bytes          [emitted]
```

```
[3Di9] ./src/print.js 62 bytes {1} [built]
[3IRH] (webpack)/buildin/module.js 517 bytes {0} [built]
[DuR2] (webpack)/buildin/global.js 509 bytes {0} [built]
   [0] multi lodash 28 bytes {0} [built]
[lVK7] ./src/index.js 421 bytes {1} [built]
    + 1 hidden module
```

And let's modify our `src/index.js` to temporarily remove that extra dependency:

src/index.js

```
  import _ from 'lodash';
- import Print from './print';
+ // import Print from './print';

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

    // Lodash, now imported by this script
    element.innerHTML = _.join(['Hello', 'webpack'], ' ');
-   element.onclick = Print.bind(null, 'Hello webpack!');
+   // element.onclick = Print.bind(null, 'Hello webpack!');

    return element;
  }

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

And finally run our build again:

```
Hash: 37e1358f135c0b992f72
Version: webpack 3.3.0
Time: 1557ms
                          Asset       Size  Chunks                    Chunk Names
  vendor.eed6dcc3b30cfa138aaa.js     541 kB       0  [emitted]  [big]  vendor
    main.fc7f38e648da79db2aba.js  891 bytes       1  [emitted]         main
manifest.bb5820632fb66c3fb357.js    5.85 kB       2  [emitted]         manifest
                     index.html  352 bytes          [emitted]
[3IRH] (webpack)/buildin/module.js 517 bytes {0} [built]
[DuR2] (webpack)/buildin/global.js 509 bytes {0} [built]
   [0] multi lodash 28 bytes {0} [built]
[lVK7] ./src/index.js 427 bytes {1} [built]
    + 1 hidden module
```

We can see that both builds yielded `eed6dcc3b30cfa138aaa` in the `vendor` bundle's filename.

Conclusion

Caching gets messy. Plain and simple. However the walk-through above should give you a running start to deploying consistent, cachable assets. See the *Further Reading* section below to learn more.

Aside from applications, webpack can also be used to bundle JavaScript libraries. The following guide is meant for library authors looking to streamline their bundling strategy.

Authoring a Library

Let's assume that you are writing a small library , `webpack-numbers` , that allows users to convert the numbers 1 through 5 from their numeric representation to a textual one and vice-versa, e.g. 2 to 'two'.

The basic project structure may look like this:

project

```
+  |- webpack.config.js
+  |- package.json
+  |- /src
+    |- index.js
+    |- ref.json
```

Initialize npm, install webpack and lodash:

```
npm init -y
npm install --save-dev webpack lodash
```

src/ref.json

```
[{
  "num": 1,
  "word": "One"
}, {
  "num": 2,
  "word": "Two"
}, {
  "num": 3,
  "word": "Three"
}, {
  "num": 4,
  "word": "Four"
}, {
  "num": 5,
  "word": "Five"
}, {
  "num": 0,
  "word": "Zero"
}]
```

src/index.js

```
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);
```

```
};
```

The usage specification for the library use will be as follows:

```
// ES2015 module import
import * as webpackNumbers from 'webpack-numbers';
// CommonJS module require
var webpackNumbers = require('webpack-numbers');
// ...
// ES2015 and CommonJS module use
webpackNumbers.wordToNum('Two');
// ...
// AMD module require
require(['webpackNumbers'], function ( webpackNumbers) {
  // ...
  // AMD module use
  webpackNumbers.wordToNum('Two');
  // ...
});
```

The consumer also can use the library by loading it via a script tag:

```
<!doctype html>
<html>
  ...
  <script src="https://unpkg.com/webpack-numbers"></script>
  <script>
    // ...
    // Global variable
    webpackNumbers.wordToNum('Five')
    // Property in the window object
    window.webpackNumbers.wordToNum('Five')
    // ...
  </script>
</html>
```

Note that we can also configure it to expose the library in the following ways:

- Property in the global object, for node.
- Property in the `this` object.

For full library configuration and code please refer to webpack-library-example.

Base Configuration

Now let's bundle this library in a way that will achieve the following goals:

- Without bundling `lodash`, but requiring it to be loaded by the consumer using `externals`.
- Setting the library name as `webpack-numbers`.
- Exposing the library as a variable called `webpackNumbers`.
- Being able to access the library inside Node.js.

Also, the consumer should be able to access the library the following ways:

- ES2015 module. i.e. `import webpackNumbers from 'webpack-numbers'`.
- CommonJS module. i.e. `require('webpack-numbers')`.
- Global variable when included through `script` tag.

We can start with this basic webpack configuration:

webpack.config.js

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

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

Externalize Lodash

Now, if you run `webpack`, you will find that a largish bundle is created. If you inspect the file, you'll see that lodash has been bundled along with your code. In this case, we'd prefer to treat `lodash` as a `peerDependency`. Meaning that the consumer should already have `lodash` installed. Hence you would want to give up control of this external library to the consumer of your library.

This can be done using the `externals` configuration:

webpack.config.js

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

  module.exports = {
    entry: './src/index.js',
    output: {
      path: path.resolve(__dirname, 'dist'),
      filename: 'webpack-numbers.js'
-   }
+   },
+   externals: {
+     lodash: {
+       commonjs: 'lodash',
+       commonjs2: 'lodash',
+       amd: 'lodash',
+       root: '_'
+     }
+   }
  };
```

This means that your library expects a dependency named `lodash` to be available in the consumer's environment.

T> Note that if you only plan on using your library as a dependency in another webpack bundle, you may specify `externals` as an array.

External Limitations

For libraries that use several files from a dependency:

```
import A from 'library/one';
import B from 'library/two';

// ...
```

You won't be able to exclude them from bundle by specifying `library` in the externals. You'll either need to exclude them one by one or by using a regular expression.

```
externals: [
  'library/one',
  'library/two',
  // Everything that starts with "library/"
  /^library\/.+$/
]
```

Expose the Library

For widespread use of the library, we would like it to be compatible in different environments, i.e. CommonJS, AMD, Node.js and as a global variable. To make your library available for consumption, add the `library` property inside `output`:

webpack.config.js

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

  module.exports = {
    entry: './src/index.js',
    output: {
      path: path.resolve(__dirname, 'dist'),
-     filename: 'webpack-numbers.js'
+     filename: 'webpack-numbers.js',
+     library: 'webpackNumbers'
    },
    externals: {
      lodash: {
        commonjs: 'lodash',
        commonjs2: 'lodash',
        amd: 'lodash',
        root: '_'
      }
    }
  };
```

T> Note that the `library` setup is tied to the `entry` configuration. For most libraries, specifying a single entry point is sufficient. While multi-part libraries are possible, it is simpler to expose partial exports through an index script that serves as a single entry point. Using an `array` as an `entry` point for a library is **not recommended**.

This exposes your library bundle available as a global variable named `webpackNumbers` when imported. To make the library compatible with other environments, add `libraryTarget` property to the config. This will add the different options about how the library can be exposed.

webpack.config.js

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

  module.exports = {
    entry: './src/index.js',
    output: {
      path: path.resolve(__dirname, 'dist'),
      filename: 'webpack-numbers.js',
-     library: 'webpackNumbers'
+     library: 'webpackNumbers',
+     libraryTarget: 'umd'
    },
    externals: {
      lodash: {
        commonjs: 'lodash',
        commonjs2: 'lodash',
        amd: 'lodash',
```

```
        root: '_'
      }
    }
  };
```

You can expose the library in the following ways:

- Variable: as a global variable made available by a `script` tag (`libraryTarget:'var'`).
- This: available through the `this` object (`libraryTarget:'this'`).
- Window: available trough the `window` object, in the browser (`libraryTarget:'window'`).
- UMD: available after AMD or CommonJS `require` (`libraryTarget:'umd'`).

If `library` is set and `libraryTarget` is not, `libraryTarget` defaults to `var` as specified in the output configuration documentation. See `output.libraryTarget` there for a detailed list of all available options.

W> With webpack 3.5.5, using `libraryTarget: { root:'_' }` doesn't work properly (as stated in issue 4824). However, you can set `libraryTarget: { var: '_' }` to expect the library as a global variable.

Final Steps

Optimize your output for production by following the steps in the production guide. Let's also add the path to your generated bundle as the package's `main` field in with our `package.json`

package.json

```
{
  ...
  "main": "dist/webpack-numbers.js",
  ...
}
```

Or, to add as standard module as per this guide:

```
{
  ...
  "module": "src/index.js",
  ...
}
```

The key `main` refers to the standard from `package.json`, and `module` to a proposal to allow the JavaScript ecosystem upgrade to use ES2015 modules without breaking backwards compatibility.

W> The `module` property should point to a script that utilizes ES2015 module syntax but no other syntax features that aren't yet supported by browsers or node. This enables webpack to parse the module syntax itself, allowing for lighter bundles via tree shaking if users are only consuming certain parts of the library.

Now you can publish it as an npm package and find it at unpkg.com to distribute it to your users.

T> To expose stylesheets associated with your library, the `ExtractTextPlugin` should be used. Users can then consume and load these as they would any other stylesheet.

The `webpack` compiler can understand modules written as ES2015 modules, CommonJS or AMD. However, some third party libraries may expect global dependencies (e.g. `$` for `jQuery`). The libraries might also create globals which need to be exported. These "broken modules" are one instance where *shimming* comes into play.

W> **We don't recommend using globals!** The whole concept behind webpack is to allow more modular front-end development. This means writing isolated modules that are well contained and do not rely on hidden dependencies (e.g. globals). Please use these features only when necessary.

Another instance where *shimming* can be useful is when you want to polyfill browser functionality to support more users. In this case, you may only want to deliver those polyfills to the browsers that need patching (i.e. load them on demand).

The following article will walk through both of these use cases.

T> For simplicity, this guide stems from the examples in Getting Started. Please make sure you are familiar with the setup there before moving on.

Shimming Globals

Let's start with the first use case of shimming global variables. Before we do anything let's take another look at our project:

project

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

Remember that `lodash` package we were using? For demonstration purposes, let's say we wanted to instead provide this as a global throughout our application. To do this, we can use the `ProvidePlugin`.

The `ProvidePlugin` makes a package available as a variable in every module compiled through webpack. If webpack sees that variable used, it will include the given package in the final bundle. Let's go ahead by removing the `import` statement for `lodash` and instead providing it via the plugin:

src/index.js

```
- import _ from 'lodash';
-
  function component() {
    var element = document.createElement('div');

-   // Lodash, now imported by this script
    element.innerHTML = _.join(['Hello', 'webpack'], ' ');

    return element;
  }

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

webpack.config.js

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

```
  module.exports = {
    entry: './src/index.js',
    output: {
      filename: 'bundle.js',
      path: path.resolve(__dirname, 'dist')
-   }
+   },
+   plugins: [
+     new webpack.ProvidePlugin({
+       _: 'lodash'
+     })
+   ]
  };
```

What we've essentially done here is tell webpack...

> If you encounter at least one instance of the variable `lodash`, include the `lodash` package and provide it to the modules that need it.

If we run a build, we should still see the same output:

```
Hash: f450fa59fa951c68c416
Version: webpack 2.2.0
Time: 343ms
    Asset    Size  Chunks                    Chunk Names
bundle.js  544 kB       0  [emitted]  [big]  main
   [0] ./~/lodash/lodash.js 540 kB {0} [built]
   [1] (webpack)/buildin/global.js 509 bytes {0} [built]
   [2] (webpack)/buildin/module.js 517 bytes {0} [built]
   [3] ./src/index.js 189 bytes {0} [built]
```

We can also use the `ProvidePlugin` to expose a single export of a module by configuring it with an "array path" (e.g. `[module, child, ...children?]`). So let's imagine we only wanted to provide the `join` method from `lodash` wherever it's invoked:

src/index.js

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

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

    return element;
  }

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

webpack.config.js

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

  module.exports = {
    entry: './src/index.js',
    output: {
      filename: 'bundle.js',
      path: path.resolve(__dirname, 'dist')
    },
    plugins: [
      new webpack.ProvidePlugin({
-       _: 'lodash'
```

```
+       join: ['lodash', 'join']
      })
    ]
  };
```

This would go nicely with Tree Shaking as the rest of the `lodash` library should get dropped.

Granular Shimming

Some legacy modules rely on `this` being the `window` object. Let's update our `index.js` so this is the case:

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

    element.innerHTML = join(['Hello', 'webpack'], ' ');
+
+   // Assume we are in the context of `window`
+   this.alert('Hmmm, this probably isn\'t a great idea...')

    return element;
  }

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

This becomes a problem when the module is executed in a CommonJS context where `this` is equal to `module.exports`. In this case you can override `this` using the `imports-loader`:

webpack.config.js

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

  module.exports = {
    entry: './src/index.js',
    output: {
      filename: 'bundle.js',
      path: path.resolve(__dirname, 'dist')
    },
+   module: {
+     rules: [
+       {
+         test: require.resolve('index.js'),
+         use: 'imports-loader?this=>window'
+       }
+     ]
+   },
    plugins: [
      new webpack.ProvidePlugin({
        join: ['lodash', 'join']
      })
    ]
  };
```

Global Exports

Let's say a library creates a global variable that it expects its consumers to use. We can add a small module to our setup to demonstrate this:

project

```
  webpack-demo
  |- package.json
  |- webpack.config.js
  |- /dist
  |- /src
    |- index.js
+   |- globals.js
  |- /node_modules
```

src/globals.js

```
var file = 'blah.txt';
var helpers = {
  test: function() { console.log('test something'); },
  parse: function() { console.log('parse something'); }
}
```

Now, while you'd likely never do this in your own source code, you may encounter a dated library you'd like to use that contains similar code to what's shown above. In this case, we can use `exports-loader`, to export that global variable as a normal module export. For instance, in order to export `file` as `file` and `helpers.parse` as `parse`:

webpack.config.js

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

  module.exports = {
    entry: './src/index.js',
    output: {
      filename: 'bundle.js',
      path: path.resolve(__dirname, 'dist')
    },
    module: {
      rules: [
        {
          test: require.resolve('index.js'),
          use: 'imports-loader?this=>window'
-       }
+       },
+       {
+         test: require.resolve('globals.js'),
+         use: 'exports-loader?file,parse=helpers.parse'
+       }
      ]
    },
    plugins: [
      new webpack.ProvidePlugin({
        join: ['lodash', 'join']
      })
    ]
  };
```

Now from within our entry script (i.e. `src/index.js`), we could `import { file, parse } from './globals.js';` and all should work smoothly.

Loading Polyfills

Almost everything we've discussed thus far has been in relation to handling legacy packages. Let's move on to our second topic: **polyfills**.

There's a lot of ways to load polyfills. For example, to include the `babel-polyfill` we might simply:

```
npm install --save babel-polyfill
```

and `import` it so as to include it in our main bundle:

src/index.js

```
+ import 'babel-polyfill';
+
  function component() {
    var element = document.createElement('div');

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

    return element;
  }

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

T> Note that we aren't binding the `import` to a variable. This is because polyfills simply run on their own, prior to the rest of the code base, allowing us to then assume certain native functionality exists.

Now while this is one approach, **including polyfills in the main bundle is not recommended** because this penalizes modern browsers users by making them download a bigger file with unneeded scripts.

Let's move our `import` to a new file and add the `whatwg-fetch` polyfill:

```
npm install --save whatwg-fetch
```

src/index.js

```
- import 'babel-polyfill';
-
  function component() {
    var element = document.createElement('div');

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

    return element;
  }

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

project

```
  webpack-demo
  |- package.json
  |- webpack.config.js
  |- /dist
  |- /src
    |- index.js
    |- globals.js
+   |- polyfills.js
  |- /node_modules
```

src/polyfills.js

```
import 'babel-polyfill';
import 'whatwg-fetch';
```

webpack.config.js

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

  module.exports = {
-   entry: './src/index.js',
+   entry: {
+     polyfills: './src/polyfills.js',
+     index: './src/index.js'
+   },
    output: {
-     filename: 'bundle.js',
+     filename: '[name].bundle.js',
      path: path.resolve(__dirname, 'dist')
    },
    module: {
      rules: [
        {
          test: require.resolve('index.js'),
          use: 'imports-loader?this=>window'
        },
        {
          test: require.resolve('globals.js'),
          use: 'exports-loader?file,parse=helpers.parse'
        }
      ]
    },
    plugins: [
      new webpack.ProvidePlugin({
        join: ['lodash', 'join']
      })
    ]
  };
```

With that in place, we can add the logic to conditionally load our new `polyfills.bundle.js` file. How you make this decision depends on the technologies and browsers you need to support. We'll just do some simple testing to determine whether our polyfills are needed:

dist/index.html

```
  <!doctype html>
  <html>
    <head>
      <title>Getting Started</title>
+     <script>
+       var modernBrowser = (
+         'fetch' in window &&
+         'assign' in Object
+       );
+
+       if ( !modernBrowser ) {
+         var scriptElement = document.createElement('script');
+
+         scriptElement.async = false;
+         scriptElement.src = '/polyfills.bundle.js';
+         document.head.appendChild(scriptElement);
+       }
+     </script>
    </head>
    <body>
      <script src="index.bundle.js"></script>
    </body>
  </html>
```

Now we can `fetch` some data within our entry script:

src/index.js

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

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

    return element;
  }

  document.body.appendChild(component());
+
+ fetch('https://jsonplaceholder.typicode.com/users')
+   .then(response => response.json())
+   .then(json => {
+     console.log('We retrieved some data! AND we\'re confident it will work on a variety of browser distributi
ons.')
+     console.log(json)
+   })
+   .catch(error => console.error('Something went wrong when fetching this data: ', error))
```

If we run our build, another `polyfills.bundle.js` file will be emitted and everything should still run smoothly in the browser. Note that this set up could likely be improved upon but it should give you a good idea of how you can provide polyfills only to the users that actually need them.

Further Optimizations

The `babel-preset-env` package uses browserslist to transpile only what is not supported in your browsers matrix. This preset comes with the `useBuiltIns` option, `false` by default, which converts your global `babel-polyfill` import to a more granular feature by feature `import` pattern:

```
import 'core-js/modules/es7.string.pad-start';
import 'core-js/modules/es7.string.pad-end';
import 'core-js/modules/web.timers';
import 'core-js/modules/web.immediate';
import 'core-js/modules/web.dom.iterable';
```

See the repository for more information.

Node Built-Ins

Node built-ins, like `process`, can be polyfilled right directly from your configuration file without the use of any special loaders or plugins. See the node configuration page for more information and examples.

Other Utilities

There are a few other tools that can help when dealing with legacy modules.

The `script-loader` evaluates code in the global context, similar to inclusion via a `script` tag. In this mode, every normal library should work. `require`, `module`, etc. are undefined.

W> When using the `script-loader`, the module is added as a string to the bundle. It is not minimized by `webpack`, so use a minimized version. There is also no `devtool` support for libraries added by this loader.

When there is no AMD/CommonJS version of the module and you want to include the `dist`, you can flag this module in `noParse`. This will cause webpack to include the module without parsing it or resolving `require()` and `import` statements. This practice is also used to improve the build performance.

W> Any feature requiring the AST, like the `ProvidePlugin`, will not work.

Lastly, there are some modules that support different module styles like AMD, CommonJS and legacy. In most of these cases, they first check for `define` and then use some quirky code to export properties. In these cases, it could help to force the CommonJS path by setting `define=>false` via the `imports-loader`.

T> This guide extends on code examples found in the Output Management guide.

Progressive Web Applications (or PWAs) are web apps that deliver an experience similar to native applications. There are many things that can contribute to that. Of these, the most significant is the ability for an app to be able to function when **offline**. This is achieved through the use of a web technology called Service Workers.

This section will focus on adding an offline experience to our app. We'll achieve this using a Google project called Workbox which provides tools that help make offline support for web apps easier to setup.

We Don't Work Offline Now

So far, we've been viewing the output by going directly to the local file system. Typically though, a real user accesses a web app over a network; their browser talking to a **server** which will serve up the required assets (e.g. `.html`, `.js`, and `.css` files).

So let's test what the current experience is like using a simple server. Let's use the http-server package: `npm install http-server --save-dev`. We'll also amend the `scripts` section of our `package.json` to add in a `start` script:

package.json

```
{
  ...
  "scripts": {
-   "build": "webpack"
+   "build": "webpack",
+   "start": "http-server dist"
  },
  ...
}
```

If you haven't previously done so, run the command `npm run build` to build your project. Then run the command `npm start`. This should produce the following output:

```
> http-server dist

Starting up http-server, serving dist
Available on:
  http://xx.x.x.x:8080
  http://127.0.0.1:8080
  http://xxx.xxx.x.x:8080
Hit CTRL-C to stop the server
```

If you open your browser to `http://localhost:8080` (i.e. `http://127.0.0.1`) you should see your webpack application being served up from the `dist` directory. If you stop the server and refresh, the webpack application is no longer available.

This is what we aim to change. Once we reach the end of this module we should be able to stop the server, hit refresh and still see our application.

Adding Workbox

Let's add the Workbox webpack plugin and adjust the `webpack.config.js` file:

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

webpack.config.js

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

  module.exports = {
    entry: {
      app: './src/index.js',
      print: './src/print.js'
    },
  plugins: [
    new CleanWebpackPlugin(['dist']),
    new HtmlWebpackPlugin({
-     title: 'Output Management'
+     title: 'Progressive Web Application'
-   })
+   }),
+   new WorkboxPlugin.GenerateSW({
+     // these options encourage the ServiceWorkers to get in there fast
+     // and not allow any straggling "old" SWs to hang around
+     clientsClaim: true,
+     skipWaiting: true
+   })
  ],
    output: {
      filename: '[name].bundle.js',
      path: path.resolve(__dirname, 'dist')
    }
  };
```

With that in place, let's see what happens when we do an `npm run build`:

```
clean-webpack-plugin: /mnt/c/Source/webpack-follow-along/dist has been removed.
Hash: 6588e31715d9be04be25
Version: webpack 3.10.0
Time: 782ms
                                          Asset       Size  Chunks                    Chunk Names
                                  app.bundle.js     545 kB    0, 1  [emitted]  [big]  app
                                print.bundle.js    2.74 kB       1  [emitted]         print
                                     index.html  254 bytes          [emitted]
precache-manifest.b5ca1c555e832d6fbf9462efd29d27eb.js  268 bytes          [emitted]
                                        sw.js       1 kB          [emitted]
   [0] ./src/print.js 87 bytes {0} {1} [built]
   [1] ./src/index.js 477 bytes {0} [built]
   [3] (webpack)/buildin/global.js 509 bytes {0} [built]
   [4] (webpack)/buildin/module.js 517 bytes {0} [built]
    + 1 hidden module
Child html-webpack-plugin for "index.html":
     1 asset
       [2] (webpack)/buildin/global.js 509 bytes {0} [built]
       [3] (webpack)/buildin/module.js 517 bytes {0} [built]
        + 2 hidden modules
```

As you can see, we now have 2 extra files being generated; `sw.js` and the more verbose `precache-manifest.b5ca1c555e832d6fbf9462efd29d27eb.js`. `sw.js` is the Service Worker file and `precache-manifest.b5ca1c555e832d6fbf9462efd29d27eb.js` is a file that `sw.js` requires so it can run. Your own generated files will likely be different; but you should have an `sw.js` file there.

So we're now at the happy point of having produced a Service Worker. What's next?

Registering Our Service Worker

Let's allow our Service Worker to come out and play by registering it. We'll do that by adding the registration code below:

index.js

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

+ if ('serviceWorker' in navigator) {
+   window.addEventListener('load', () => {
+     navigator.serviceWorker.register('/sw.js').then(registration => {
+       console.log('SW registered: ', registration);
+     }).catch(registrationError => {
+       console.log('SW registration failed: ', registrationError);
+     });
+   });
+ }
```

Once more `npm run build` to build a version of the app including the registration code. Then serve it with `npm start`. Navigate to `http://localhost:8080` and take a look at the console. Somewhere in there you should see:

```
SW registered
```

Now to test it. Stop your server and refresh your page. If your browser supports Service Workers then you should still be looking at your application. However, it has been served up by your Service Worker and **not** by the server.

Conclusion

You have built an offline app using the Workbox project. You've started the journey of turning your web app into a PWA. You may now want to think about taking things further. A good resource to help you with that can be found here.

T> This guide stems from the *Getting Started* guide.

TypeScript is a typed superset of JavaScript that compiles to plain JavaScript. In this guide we will learn how to integrate TypeScript with webpack.

Basic Setup

First install the TypeScript compiler and loader by running:

```
npm install --save-dev typescript ts-loader
```

Now we'll modify the directory structure & the configuration files:

project

```
  webpack-demo
  |- package.json
+ |- tsconfig.json
  |- webpack.config.js
  |- /dist
    |- bundle.js
    |- index.html
  |- /src
    |- index.js
+   |- index.ts
  |- /node_modules
```

tsconfig.json

Let's set up a simple configuration to support JSX and compile TypeScript down to ES5...

```
{
  "compilerOptions": {
    "outDir": "./dist/",
    "noImplicitAny": true,
    "module": "es6",
    "target": "es5",
    "jsx": "react",
    "allowJs": true
  }
}
```

See TypeScript's documentation to learn more about `tsconfig.json` configuration options.

To learn more about webpack configuration, see the configuration concepts.

Now let's configure webpack to handle TypeScript:

webpack.config.js

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

module.exports = {
  entry: './src/index.ts',
  module: {
    rules: [
      {
        test: /\.tsx?$/,
        use: 'ts-loader',
        exclude: /node_modules/
      }
```

```
    ]
  },
  resolve: {
    extensions: [ '.tsx', '.ts', '.js' ]
  },
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist')
  }
};
```

This will direct webpack to *enter* through `./index.ts`, *load* all `.ts` and `.tsx` files through the `ts-loader`, and *output* a `bundle.js` file in our current directory.

Loader

`ts-loader`

We use `ts-loader` in this guide as it makes enabling additional webpack features, such as importing other web assets, a bit easier.

Source Maps

To learn more about source maps, see the development guide.

To enable source maps, we must configure TypeScript to output inline source maps to our compiled JavaScript files. The following line must be added to our TypeScript configuration:

tsconfig.json

```
  {
    "compilerOptions": {
      "outDir": "./dist/",
+     "sourceMap": true,
      "noImplicitAny": true,
      "module": "commonjs",
      "target": "es5",
      "jsx": "react",
      "allowJs": true
    }
  }
```

Now we need to tell webpack to extract these source maps and into our final bundle:

webpack.config.js

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

  module.exports = {
    entry: './src/index.ts',
+   devtool: 'inline-source-map',
    module: {
      rules: [
        {
          test: /\.tsx?$/,
          use: 'ts-loader',
          exclude: /node_modules/
        }
      ]
    },
```

```
    resolve: {
      extensions: [ '.tsx', '.ts', '.js' ]
    },
    output: {
      filename: 'bundle.js',
      path: path.resolve(__dirname, 'dist')
    }
  };
```

See the devtool documentation for more information.

Using Third Party Libraries

When installing third party libraries from npm, it is important to remember to install the typing definition for that library. These definitions can be found at TypeSearch.

For example if we want to install lodash we can run the following command to get the typings for it:

```
npm install --save-dev @types/lodash
```

For more information see this blog post.

Importing Other Assets

To use non-code assets with TypeScript, we need to defer the type for these imports. This requires a `custom.d.ts` file which signifies custom definitions for TypeScript in our project. Let's set up a declaration for `.svg` files:

custom.d.ts

```
declare module "*.svg" {
  const content: any;
  export default content;
}
```

Here we declare a new module for SVGs by specifying any import that ends in `.svg` and defining the module's `content` as `any`. We could be more explicit about it being a url by defining the type as string. The same concept applies to other assets including CSS, SCSS, JSON and more.

Build Performance

W> This may degrade build performance.

See the Build Performance guide on build tooling.

The following sections describe the major changes from webpack 1 to 2.

T> Note that there were far fewer changes between 2 and 3, so that migration shouldn't be too bad. If you are running into issues, please see the changelog for details.

W> This content may be moved to the blog post in the near future as version 2 has been out for a while. On top of that, version 3 was recently released and version 4 is on the horizon. As noted above, folks should instead to refer to the changelog for migrations.

`resolve.root`, `resolve.fallback`, `resolve.modulesDirectories`

These options were replaced by a single option `resolve.modules`. See resolving for more usage.

```
  resolve: {
-   root: path.join(__dirname, "src")
+   modules: [
+     path.join(__dirname, "src"),
+     "node_modules"
+   ]
  }
```

`resolve.extensions`

This option no longer requires passing an empty string. This behavior was moved to `resolve.enforceExtension`. See resolving for more usage.

`resolve.*`

Several APIs were changed here. Not listed in detail as it's not commonly used. See resolving for details.

`module.loaders` is now `module.rules`

The old loader configuration was superseded by a more powerful rules system, which allows configuration of loaders and more. For compatibility reasons, the old `module.loaders` syntax is still valid and the old names are parsed. The new naming conventions are easier to understand and are a good reason to upgrade the configuration to using `module.rules`.

```
  module: {
-   loaders: [
+   rules: [
      {
        test: /\.css$/,
-       loaders: [
-         "style-loader",
-         "css-loader?modules=true"
+       use: [
+         {
+           loader: "style-loader"
+         },
+         {
+           loader: "css-loader",
+           options: {
+             modules: true
+           }
```

```
+         }
        ]
      },
      {
        test: /\.jsx$/,
        loader: "babel-loader", // Do not use "use" here
        options: {
          // ...
        }
      }
    ]
  }
```

Chaining loaders

Like in webpack 1, loaders can be chained to pass results from loader to loader. Using the rule.use configuration option, `use` can be set to an array of loaders. In webpack 1, loaders were commonly chained with `!`. This style is only supported using the legacy option `module.loaders`.

```
  module: {
-   loaders: [{
+   rules: [{
      test: /\.less$/,
-     loader: "style-loader!css-loader!less-loader"
+     use: [
+       "style-loader",
+       "css-loader",
+       "less-loader"
+     ]
    }]
  }
```

Automatic `-loader` module name extension removed

It is not possible anymore to omit the `-loader` extension when referencing loaders:

```
  module: {
    rules: [
      {
        use: [
-         "style",
+         "style-loader",
-         "css",
+         "css-loader",
-         "less",
+         "less-loader",
        ]
      }
    ]
  }
```

You can still opt-in to the old behavior with the `resolveLoader.moduleExtensions` configuration option, but this is not recommended.

```
+ resolveLoader: {
+   moduleExtensions: ["-loader"]
+ }
```

See #2986 for the reason behind this change.

`json-loader` is not required anymore

When no loader has been configured for a JSON file, webpack will automatically try to load the JSON file with the `json-loader`.

```
  module: {
    rules: [
-     {
-       test: /\.json/,
-       loader: "json-loader"
-     }
    ]
  }
```

We decided to do this in order to iron out environment differences between webpack, node.js and browserify.

Loaders in configuration resolve relative to context

In **webpack 1**, configured loaders resolve relative to the matched file. However, in **webpack 2**, configured loaders resolve relative to the `context` option.

This solves some problems with duplicate modules caused by loaders when using `npm link` or referencing modules outside of the `context`.

You may remove some hacks to work around this:

```
  module: {
    rules: [
      {
        // ...
-       loader: require.resolve("my-loader")
+       loader: "my-loader"
      }
    ]
  },
  resolveLoader: {
-   root: path.resolve(__dirname, "node_modules")
  }
```

`module.preLoaders` and `module.postLoaders` were removed:

```
  module: {
-   preLoaders: [
+   rules: [
      {
        test: /\.js$/,
+       enforce: "pre",
        loader: "eslint-loader"
      }
    ]
  }
```

`UglifyJsPlugin` sourceMap

The `sourceMap` option of the `UglifyJsPlugin` now defaults to `false` instead of `true`. This means that if you are using source maps for minimized code or want correct line numbers for uglifyjs warnings, you need to set `sourceMap: true` for `UglifyJsPlugin`.

```
  devtool: "source-map",
  plugins: [
    new UglifyJsPlugin({
+     sourceMap: true
    })
  ]
```

`UglifyJsPlugin` warnings

The `compress.warnings` option of the `UglifyJsPlugin` now defaults to `false` instead of `true`. This means that if you want to see uglifyjs warnings, you need to set `compress.warnings` to `true`.

```
  devtool: "source-map",
  plugins: [
    new UglifyJsPlugin({
+     compress: {
+       warnings: true
+     }
    })
  ]
```

`UglifyJsPlugin` minimize loaders

`UglifyJsPlugin` no longer switches loaders into minimize mode. The `minimize: true` setting needs to be passed via loader options in the long-term. See loader documentation for relevant options.

The minimize mode for loaders will be removed in webpack 3 or later.

To keep compatibility with old loaders, loaders can be switched to minimize mode via plugin:

```
  plugins: [
+   new webpack.LoaderOptionsPlugin({
+     minimize: true
+   })
  ]
```

`DedupePlugin` has been removed

`webpack.optimize.DedupePlugin` isn't needed anymore. Remove it from your configuration.

`BannerPlugin` - breaking change

`BannerPlugin` no longer accepts two parameters, but a single options object.

```
  plugins: [
-    new webpack.BannerPlugin('Banner', {raw: true, entryOnly: true});
+    new webpack.BannerPlugin({banner: 'Banner', raw: true, entryOnly: true});
  ]
```

`OccurrenceOrderPlugin` is now on by default

The `OccurrenceOrderPlugin` is now enabled by default and has been renamed (`OccurenceOrderPlugin` in webpack 1). Thus make sure to remove the plugin from your configuration:

```
  plugins: [
    // webpack 1
-   new webpack.optimize.OccurenceOrderPlugin()
    // webpack 2
-   new webpack.optimize.OccurrenceOrderPlugin()
  ]
```

`ExtractTextWebpackPlugin` - breaking change

ExtractTextPlugin requires version 2 to work with webpack 2.

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

The configuration changes for this plugin are mainly syntactical.

`ExtractTextPlugin.extract`

```
module: {
  rules: [
    {
      test: /.css$/,
-      loader: ExtractTextPlugin.extract("style-loader", "css-loader", { publicPath: "/dist" })
+      use: ExtractTextPlugin.extract({
+        fallback: "style-loader",
+        use: "css-loader",
+        publicPath: "/dist"
+      })
    }
  ]
}
```

`new ExtractTextPlugin({options})`

```
plugins: [
-  new ExtractTextPlugin("bundle.css", { allChunks: true, disable: false })
+  new ExtractTextPlugin({
+    filename: "bundle.css",
+    disable: false,
+    allChunks: true
+  })
]
```

Full dynamic requires now fail by default

A dependency with only an expression (i. e. `require(expr)`) will now create an empty context instead of the context of the complete directory.

Code like this should be refactored as it won't work with ES2015 modules. If this is not possible you can use the `ContextReplacementPlugin` to hint the compiler towards the correct resolving.

?> Link to an article about dynamic dependencies.

Using custom arguments in CLI and configuration

If you abused the CLI to pass custom arguments to the configuration like so:

`webpack --custom-stuff`

```
// webpack.config.js
var customStuff = process.argv.indexOf("--custom-stuff") >= 0;
/* ... */
module.exports = config;
```

You may notice that this is no longer allowed. The CLI is more strict now.

Instead there is an interface for passing arguments to the configuration. This should be used instead. Future tools may rely on this.

`webpack --env.customStuff`

```
module.exports = function(env) {
  var customStuff = env.customStuff;
  /* ... */
  return config;
};
```

See CLI.

`require.ensure` and AMD `require` are asynchronous

These functions are now always asynchronous instead of calling their callback synchronously if the chunk is already loaded.

`require.ensure` now depends upon native `Promise`s. If using `require.ensure` in an environment that lacks them then you will need a polyfill.

Loader configuration is through `options`

You can *no longer* configure a loader with a custom property in the `webpack.config.js`. It must be done through the `options`. The following configuration with the `ts` property is no longer valid with webpack 2:

```
module.exports = {
  ...
  module: {
    rules: [{
      test: /\.tsx?$/,
      loader: 'ts-loader'
    }]
  },
  // does not work with webpack 2
  ts: { transpileOnly: false }
}
```

What are `options`?

Good question. Well, strictly speaking it's 2 possible things; both ways to configure a webpack loader. Classically `options` was called `query` and was a string which could be appended to the name of the loader. Much like a query string but actually with greater powers:

```
module.exports = {
  ...
  module: {
    rules: [{
      test: /\.tsx?$/,
      loader: 'ts-loader?' + JSON.stringify({ transpileOnly: false })
    }]
  }
}
```

But it can also be a separately specified object that's supplied alongside a loader:

```
module.exports = {
  ...
  module: {
    rules: [{
      test: /\.tsx?$/,
      loader: 'ts-loader',
      options:  { transpileOnly: false }
    }]
  }
}
```

`LoaderOptionsPlugin` context

Some loaders need context information and read them from the configuration. This needs to be passed via loader options in the long-term. See loader documentation for relevant options.

To keep compatibility with old loaders, this information can be passed via plugin:

```
  plugins: [
+   new webpack.LoaderOptionsPlugin({
+     options: {
+       context: __dirname
+     }
+   })
  ]
```

`debug`

The `debug` option switched loaders to debug mode in webpack 1. This needs to be passed via loader options in long-term. See loader documentation for relevant options.

The debug mode for loaders will be removed in webpack 3 or later.

To keep compatibility with old loaders, loaders can be switched to debug mode via a plugin:

```
- debug: true,
  plugins: [
+   new webpack.LoaderOptionsPlugin({
+     debug: true
+   })
  ]
```

Code Splitting with ES2015

In webpack 1, you could use `require.ensure()` as a method to lazily-load chunks for your application:

```
require.ensure([], function(require) {
  var foo = require("./module");
});
```

The ES2015 Loader spec defines `import()` as method to load ES2015 Modules dynamically on runtime. webpack treats `import()` as a split-point and puts the requested module in a separate chunk. `import()` takes the module name as argument and returns a Promise.

```
function onClick() {
  import("./module").then(module => {
    return module.default;
  }).catch(err => {
    console.log("Chunk loading failed");
  });
}
```

Good news: Failure to load a chunk can now be handled because they are `Promise` based.

Dynamic expressions

It's possible to pass a partial expression to `import()`. This is handled similar to expressions in CommonJS (webpack creates a context with all possible files).

`import()` creates a separate chunk for each possible module.

```
function route(path, query) {
  return import(`./routes/${path}/route`)
    .then(route => new route.Route(query));
}
// This creates a separate chunk for each possible route
```

Mixing ES2015 with AMD and CommonJS

As for AMD and CommonJS you can freely mix all three module types (even within the same file). webpack behaves similar to babel and node-eps in this case:

```
// CommonJS consuming ES2015 Module
var book = require("./book");

book.currentPage;
book.readPage();
book.default === "This is a book";
```

```
// ES2015 Module consuming CommonJS
import fs from "fs"; // module.exports map to default
import { readFileSync } from "fs"; // named exports are read from returned object+

typeof fs.readFileSync === "function";
typeof readFileSync === "function";
```

It is important to note that you will want to tell Babel to not parse these module symbols so webpack can use them. You can do this by setting the following in your `.babelrc` or `babel-loader` options.

.babelrc

```
{
  "presets": [
    ["es2015", { "modules": false }]
  ]
}
```

Hints

No need to change something, but opportunities

Template strings

webpack now supports template strings in expressions. This means you can start using them in webpack constructs:

```
- require("./templates/" + name);
+ require(`./templates/${name}`);
```

Configuration Promise

webpack now supports returning a `Promise` from the configuration file. This allows async processing in your configuration file.

webpack.config.js

```
module.exports = function() {
  return fetchLangs().then(lang => ({
    entry: "...",
    // ...
    plugins: [
      new DefinePlugin({ LANGUAGE: lang })
    ]
  }));
};
```

Advanced loader matching

webpack now supports more things to match on for loaders.

```
module: {
  rules: [
    {
      resource: /filename/, // matches "/path/filename.js"
      resourceQuery: /^\?querystring$/, // matches "?querystring"
      issuer: /filename/, // matches "/path/something.js" if requested from "/path/filename.js"
    }
  ]
}
```

More CLI options

There are some new CLI options for you to use:

`--define process.env.NODE_ENV="production"` See `DefinePlugin`.

`--display-depth` displays the distance to the entry point for each module.

`--display-used-exports` display info about which exports are used in a module.

`--display-max-modules` sets the number for modules displayed in the output (defaults to 15).

`-p` also defines `process.env.NODE_ENV` to `"production"` now.

Loader changes

Changes only relevant for loader authors.

Cacheable

Loaders are now cacheable by default. Loaders must opt-out if they are not cacheable.

```
  // Cacheable loader
  module.exports = function(source) {
-   this.cacheable();
    return source;
  }
```

```
  // Not cacheable loader
  module.exports = function(source) {
+   this.cacheable(false);
    return source;
  }
```

Complex options

webpack 1 only supports `JSON.stringify` -able options for loaders.

webpack 2 now supports any JS object as loader options.

Before webpack 2.2.1 (i.e. from 2.0.0 through 2.2.0), using complex options required using `ident` for the `options` object to allow its reference from other loaders. **This was removed in 2.2.1** and thus current migrations do not require any use of the `ident` key.

```
{
  test: /\.ext/
  use: {
    loader: '...',
    options: {
-     ident: 'id',
      fn: () => require('./foo.js')
    }
  }
}
```

To disambiguate in your `webpack.config.js` between development and production builds you may use environment variables.

The webpack command line environment option `--env` allows you to pass in as many environment variables as you like. Environment variables will be made accessible in your `webpack.config.js`. For example, `--env.production` or `--env.NODE_ENV=local` (`NODE_ENV` is conventionally used to define the environment type, see here.)

```
webpack --env.NODE_ENV=local --env.production --progress
```

T> Setting up your `env` variable without assignment, `--env.production` sets `--env.production` to `true` by default. There are also other syntaxes that you can use. See the webpack CLI documentation for more information.

There is one change that you will have to make to your webpack config. Typically, `module.exports` points to the configuration object. To use the `env` variable, you must convert `module.exports` to a function:

webpack.config.js

```
module.exports = env => {
  // Use env.<YOUR VARIABLE> here:
  console.log('NODE_ENV: ', env.NODE_ENV) // 'local'
  console.log('Production: ', env.production) // true

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

This guide contains some useful tips for improving build/compilation performance.

General

The following best practices should help whether or not you are in development or building for production.

Stay Up to Date

Use the latest webpack version. We are always making performance improvements. The latest stable version of webpack is:

Staying up to date with **Node.js** can also help with performance. On top of this, keeping your package manager (e.g. `npm` or `yarn`) up to date can also help. Newer versions create more efficient module trees and increase resolving speed.

Loaders

Apply loaders to the minimal number of modules necessary. Instead of:

```
{
  test: /\.js$/,
  loader: "babel-loader"
}
```

Use the `include` field to only apply the loader modules that actually need to be transformed by it:

```
{
  test: /\.js$/,
  include: path.resolve(__dirname, "src"),
  loader: "babel-loader"
}
```

Bootstrap

Each additional loader/plugin has a bootup time. Try to use as few different tools as possible.

Resolving

The following steps can increase the speed of resolving:

- Minimize the number of items in `resolve.modules`, `resolve.extensions`, `resolve.mainFiles`, `resolve.descriptionFiles` as they increase the number of filesystem calls.
- Set `resolve.symlinks: false` if you don't use symlinks (e.g. `npm link` or `yarn link`).
- Set `resolve.cacheWithContext: false` if you use custom resolving plugins, that are not context specific.

Dlls

Use the `DllPlugin` to move code that is changed less often into a separate compilation. This will improve the application's compilation speed, although it does increase complexity of the build process.

Smaller = Faster

Decrease the total size of the compilation to increase build performance. Try to keep chunks small.

- Use fewer/smaller libraries.
- Use the `CommonsChunksPlugin` in Multi-Page Applications.
- Use the `CommonsChunksPlugin` in `async` mode in Multi-Page Applications.
- Remove unused code.
- Only compile the part of the code you are currently developing on.

Worker Pool

The `thread-loader` can be used to offload expensive loaders to a worker pool.

W> Don't use too many workers as there is a boot overhead for the Node.js runtime and the loader. Minimize the module transfers between worker and main process. IPC is expensive.

Persistent cache

Enable persistent caching with the `cache-loader`. Clear cache directory on `"postinstall"` in `package.json`.

Custom plugins/loaders

Profile them to not introduce a performance problem here.

Development

The following steps are especially useful in *development*.

Incremental Builds

Use webpack's watch mode. Don't use other tools to watch your files and invoke webpack. The built in watch mode will keep track of timestamps and passes this information to the compilation for cache invalidation.

In some setups watching falls back to polling mode. With many watched files this can cause a lot of CPU load. In these cases you can increase the polling interval with `watchOptions.poll`.

Compile in Memory

The following utilities improve performance by compiling and serving assets in memory rather than writing to disk:

- `webpack-dev-server`
- `webpack-hot-middleware`
- `webpack-dev-middleware`

Devtool

Be aware of the performance differences of the different `devtool` settings.

- `"eval"` has the best performance, but doesn't assist you for transpiled code.
- The `cheap-source-map` variants are more performant, if you can live with the slightly worse mapping quality.
- Use a `eval-source-map` variant for incremental builds.

=> In most cases `cheap-module-eval-source-map` is the best option.

Avoid Production Specific Tooling

Certain utilities, plugins and loader only make sense when building for production. For example, it usually doesn't make sense to minify and mangle your code with the `UglifyJsPlugin` while in development. These tools should typically be excluded in development:

- `UglifyJsPlugin`
- `ExtractTextPlugin`
- `[hash]` / `[chunkhash]`
- `AggressiveSplittingPlugin`
- `AggressiveMergingPlugin`
- `ModuleConcatenationPlugin`

Minimal Entry Chunk

webpack only emits updated chunks to the filesystem. For some configuration options (HMR, `[name]` / `[chunkhash]` in `output.chunkFilename`, `[hash]`) the entry chunk is invalidated in addition to the changed chunks.

Make sure the entry chunk is cheap to emit by keeping it small. The following code block extracts a chunk containing only the runtime with *all other chunks as children*:

```
new CommonsChunkPlugin({
  name: "manifest",
  minChunks: Infinity
})
```

Production

The following steps are especially useful in *production*.

W> **Don't sacrifice the quality of your application for small performance gains!** Keep in mind that optimization quality is in most cases more important than build performance.

Multiple Compilations

When using multiple compilations the following tools can help:

- `parallel-webpack`: It allows to do compilation in a worker pool.
- `cache-loader`: The cache can be shared between multiple compilations.

Source Maps

Source maps are really expensive. Do you really need them?

Specific Tooling Issues

The following tools have certain problems that can degrade build performance.

Babel

- Minimize the number of preset/plugins

Typescript

- Use the `fork-ts-checker-webpack-plugin` for type checking in a separate process.
- Configure loaders to skip typechecking.
- Use the `ts-loader` in `happyPackMode: true` / `transpileOnly: true`.

Sass

- `node-sass` has a bug which blocks threads from the Node.js threadpool. When using it with the `thread-loader` set `workerParallelJobs: 2`.

Webpack is capable of adding `nonce` to all scripts that it loads. To activate the feature set a `__webpack_nonce__` variable needs to be included in your entry script. A unique hash based nonce should be generated and provided for each unique page view this is why `__webpack_nonce__` is specified in the entry file and not in the configuration. Please note that `nonce` should always be a base64-encoded string.

Examples

In the entry file:

```
// ...
__webpack_nonce__ = 'c29tZSBjb29sIHN0cmluZyB3aWxsIHBvcCB1cCAxMjM=';
// ...
```

Enabling CSP

Please note that CSPs are not enable by default. A corresponding header `Content-Security-Policy` or meta tag `<meta http-equiv="Content-Security-Policy" ...>` needs to be sent with the document to instruct the browser to enable the CSP. Here's an example of what a CSP header including a CDN white-listed URL might look like:

```
Content-Security-Policy: default-src 'self'; script-src 'self' https://trusted.cdn.com;
```

For more information on CSP and `nonce` attribute, please refer to **Further Reading** section at the bottom of this page.

If you have a more advanced project and use Vagrant to run your development environment in a Virtual Machine, you'll often want to also run webpack in the VM.

Configuring the Project

To start, make sure that the `Vagrantfile` has a static IP;

```
Vagrant.configure("2") do |config|
  config.vm.network :private_network, ip: "10.10.10.61"
end
```

Next, install webpack and webpack-dev-server in your project;

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

Make sure to have a `webpack.config.js` file. If you haven't already, use this as a minimal example to get started:

```
module.exports = {
  context: __dirname,
  entry: "./app.js"
};
```

And create a `index.html` file. The script tag should point to your bundle. If `output.filename` is not specified in the config, this will be `bundle.js` .

```
<!doctype html>
<html>
  <head>
    <script src="/bundle.js" charset="utf-8"></script>
  </head>
  <body>
    <h2>Heey!</h2>
  </body>
</html>
```

Note that you also need to create an `app.js` file.

Running the Server

Now, let's run the server:

```
webpack-dev-server --host 0.0.0.0 --public 10.10.10.61:8080 --watch-poll
```

By default the server will only be accessible from localhost. We'll be accessing it from our host PC, so we need to change `--host` to allow this.

webpack-dev-server will include a script in your bundle that connects to a WebSocket to reload when a change in any of your files occurs. The `--public` flag makes sure the script knows where to look for the WebSocket. The server will use port `8080` by default, so we should also specify that here.

`--watch-poll` makes sure that webpack can detect changes in your files. By default webpack listens to events triggered by the filesystem, but VirtualBox has many problems with this.

The server should be accessible on `http://10.10.10.61:8080` now. If you make a change in `app.js`, it should live reload.

Advanced Usage with nginx

To mimic a more production-like environment, it is also possible to proxy the webpack-dev-server with nginx.

In your nginx config file, add the following:

```
server {
  location / {
    proxy_pass http://127.0.0.1:8080;
    proxy_http_version 1.1;
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection "upgrade";
    error_page 502 @start-webpack-dev-server;
  }

  location @start-webpack-dev-server {
    default_type text/plain;
    return 502 "Please start the webpack-dev-server first.";
  }
}
```

The `proxy_set_header` lines are important, because they allow the WebSockets to work correctly.

The command to start webpack-dev-server can then be changed to this:

```
webpack-dev-server --public 10.10.10.61 --watch-poll
```

This makes the server only accessible on `127.0.0.1`, which is fine, because nginx takes care of making it available on your host PC.

Conclusion

We made the Vagrant box accessible from a static IP, and then made webpack-dev-server publicly accessible so it is reachable from a browser. We then tackled a common problem that VirtualBox doesn't send out filesystem events, causing the server to not reload on file changes.

> es6 modules
>
> commonjs
>
> amd

require with expression

A context is created if your request contains expressions, so the **exact** module is not known on compile time.

Example:

```
require("./template/" + name + ".ejs");
```

webpack parses the `require()` call and extracts some information:

```
Directory: ./template
Regular expression: /^.*\.ejs$/
```

context module

A context module is generated. It contains references to **all modules in that directory** that can be required with a request matching the regular expression. The context module contains a map which translates requests to module ids.

Example:

```
{
    "./table.ejs": 42,
    "./table-row.ejs": 43,
    "./directory/folder.ejs": 44
}
```

The context module also contains some runtime logic to access the map.

This means dynamic requires are supported but will cause all possible modules to be included in the bundle.

`require.context`

You can create your own context with the `require.context()` function.

It allows you to pass in a directory to search, a flag indicating whether subdirectories should be searched too, and a regular expression to match files against.

webpack parses for `require.context()` in the code while building.

The syntax is as follows:

```
require.context(directory, useSubdirectories = false, regExp = /^\.\//)
```

Examples:

```
require.context("./test", false, /\.test\.js$/);
// a context with files from the test directory that can be required with a request endings with `.test.js`.
```

```
require.context("../", true, /\.stories\.js$/);
// a context with all files in the parent folder and descending folders ending with `.stories.js`.
```

W> The arguments passed to `require.context` must be literals!

context module API

A context module exports a (require) function that takes one argument: the request.

The exported function has 3 properties: `resolve`, `keys`, `id`.

- `resolve` is a function and returns the module id of the parsed request.
- `keys` is a function that returns an array of all possible requests that the context module can handle.

This can be useful if you want to require all files in a directory or matching a pattern, Example:

```
function importAll (r) {
  r.keys().forEach(r);
}

importAll(require.context('../components/', true, /\.js$/));
```

```
var cache = {};

function importAll (r) {
  r.keys().forEach(key => cache[key] = r(key));
}

importAll(require.context('../components/', true, /\.js$/));
// At build-time cache will be populated with all required modules.
```

- `id` is the module id of the context module. This may be useful for `module.hot.accept`.

The `publicPath` configuration option can be quite useful in a variety of scenarios. It allows you to specify the base path for all the assets within your application.

Use Cases

There are a few use cases in real applications where this feature becomes especially neat. Essentially, every file emitted to your `output.path` directory will be referenced from the `output.publicPath` location. This includes child chunks (created via code splitting) and any other assets (e.g. images, fonts, etc.) that are a part of your dependency graph.

Environment Based

In development for example, we might have an `assets/` folder that lives on the same level of our index page. This is fine, but what if we wanted to host all these static assets on a CDN in production?

To approach this problem you can easily use a good old environment variable. Let's say we have a variable `ASSET_PATH`:

```
import webpack from 'webpack';

// Try the environment variable, otherwise use root
const ASSET_PATH = process.env.ASSET_PATH || '/';

export default {
  output: {
    publicPath: ASSET_PATH
  },

  plugins: [
    // This makes it possible for us to safely use env vars on our code
    new webpack.DefinePlugin({
      'process.env.ASSET_PATH': JSON.stringify(ASSET_PATH)
    })
  ]
};
```

On The Fly

Another possible use case is to set the `publicPath` on the fly. webpack exposes a global variable called `__webpack_public_path__` that allows you to do that. So, in your application's entry point, you can simply do this:

```
__webpack_public_path__ = process.env.ASSET_PATH;
```

That's all you need. Since we're already using the `DefinePlugin` on our configuration, `process.env.ASSET_PATH` will always be defined so we can safely do that.

W> Be aware that if you use ES6 module imports in your entry file the `__webpack_public_path__` assignment will be done after the imports. In such cases, you'll have to move the public path assignment to its own dedicated module and then import it on top of your entry.js:

```
// entry.js
import './public-path';
import './app';
```

Let's start by clearing up a common misconception. webpack is a module bundler like Browserify or Brunch. It is *not a task runner* like Make, Grunt, or Gulp. Task runners handle automation of common development tasks such as linting, building, or testing your project. Compared to bundlers, task runners have a higher level focus. You can still benefit from their higher level tooling while leaving the problem of bundling to webpack.

Bundlers help you get your JavaScript and stylesheets ready for deployment, transforming them into a format that's suitable for the browser. For example, JavaScript can be minified or split into chunks and lazy-loaded to improve performance. Bundling is one of the most important challenges in web development, and solving it well can remove a lot of pain from the process.

The good news is that, while there is some overlap, task runners and bundlers can play well together if approached in the right way. This guide provides a high-level overview of how webpack can be integrated into some of the more popular task runners.

NPM Scripts

Often webpack users use npm `scripts` as their task runner. This is a good starting point. Cross-platform support can become a problem, but there are several workarounds for that. Many, if not most users, get by with simple npm `scripts` and various levels of webpack configuration and tooling.

So while webpack's core focus is bundling, there are a variety of extensions that can enable you to use it for jobs typical of a task runner. Integrating a separate tool adds complexity, so be sure to weigh the pros and cons before going forward.

Grunt

For those using Grunt, we recommend the `grunt-webpack` package. With `grunt-webpack` you can run webpack or webpack-dev-server as a task, get access to stats within template tags, split development and production configurations and more. Start by installing `grunt-webpack` as well as `webpack` itself if you haven't already:

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

Then register a configuration and load the task:

Gruntfile.js

```
const webpackConfig = require('./webpack.config.js');

module.exports = function(grunt) {
  grunt.initConfig({
    webpack: {
      options: {
        stats: !process.env.NODE_ENV || process.env.NODE_ENV === 'development'
      },
      prod: webpackConfig,
      dev: Object.assign({ watch: true }, webpackConfig)
    }
  });

  grunt.loadNpmTasks('grunt-webpack');
};
```

For more information, please visit the repository.

Gulp

Gulp is also a fairly straightforward integration with the help of the `webpack-stream` package (a.k.a. `gulp-webpack`). In this case, it is unnecessary to install `webpack` separately as it is a direct dependency of `webpack-stream`:

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

Just `require('webpack-stream')` instead of `webpack` and optionally pass it an configuration:

gulpfile.js

```
var gulp = require('gulp');
var webpack = require('webpack-stream');
gulp.task('default', function() {
  return gulp.src('src/entry.js')
    .pipe(webpack({
      // Any configuration options...
    }))
    .pipe(gulp.dest('dist/'));
});
```

For more information, please visit the repository.

Mocha

The `mocha-webpack` utility can be used for a clean integration with Mocha. The repository offers more details on the pros and cons but essentially `mocha-webpack` is a simple wrapper that provides almost the same CLI as Mocha itself and provides various webpack functionality like an improved watch mode and improved path resolution. Here is a small example of how you would install it and use it to run a test suite (found within `./test`):

```
npm install --save-dev webpack mocha mocha-webpack
mocha-webpack 'test/**/*.js'
```

For more information, please visit the repository.

Karma

The `karma-webpack` package allows you to use webpack to pre-process files in Karma. It also makes use of `webpack-dev-middleware` and allows passing configurations for both. A simple example may look something like this:

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

karma.conf.js

```
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...
    }
  });
};
```

For more information, please visit the repository.

webpack enables use of loaders to preprocess files. This allows you to bundle any static resource way beyond JavaScript. You can easily write your own loaders using Node.js.

Loaders are activated by using `loadername!` prefixes in `require()` statements, or are automatically applied via regex from your webpack configuration – see configuration.

Files

- `raw-loader` Loads raw content of a file (utf-8)
- `val-loader` Executes code as module and consider exports as JS code
- `url-loader` Works like the file loader, but can return a data URL if the file is smaller than a limit
- `file-loader` Emits the file into the output folder and returns the (relative) URL

JSON

- `json-loader` Loads a JSON file (included by default)
- `json5-loader` Loads and transpiles a JSON 5 file
- `cson-loader` Loads and transpiles a CSON file

Transpiling

- `script-loader` Executes a JavaScript file once in global context (like in script tag), requires are not parsed
- `babel-loader` Loads ES2015+ code and transpiles to ES5 using Babel
- `buble-loader` Loads ES2015+ code and transpiles to ES5 using Bublé
- `traceur-loader` Loads ES2015+ code and transpiles to ES5 using Traceur
- `ts-loader` or `awesome-typescript-loader` Loads TypeScript 2.0+ like JavaScript
- `coffee-loader` Loads CoffeeScript like JavaScript

Templating

- `html-loader` Exports HTML as string, require references to static resources
- `pug-loader` Loads Pug templates and returns a function
- `jade-loader` Loads Jade templates and returns a function
- `markdown-loader` Compiles Markdown to HTML
- `react-markdown-loader` Compiles Markdown to a React Component using the markdown-parse parser
- `posthtml-loader` Loads and transforms a HTML file using PostHTML
- `handlebars-loader` Compiles Handlebars to HTML
- `markup-inline-loader` Inline SVG/MathML files to HTML. It's useful when applying icon font or applying CSS animation to SVG.

Styling

- `style-loader` Add exports of a module as style to DOM
- `css-loader` Loads CSS file with resolved imports and returns CSS code
- `less-loader` Loads and compiles a LESS file
- `sass-loader` Loads and compiles a SASS/SCSS file
- `postcss-loader` Loads and transforms a CSS/SSS file using PostCSS
- `stylus-loader` Loads and compiles a Stylus file

Linting && Testing

- `mocha-loader` Tests with mocha (Browser/NodeJS)
- `eslint-loader` PreLoader for linting code using ESLint
- `jshint-loader` PreLoader for linting code using JSHint
- `jscs-loader` PreLoader for code style checking using JSCS
- `coverjs-loader` PreLoader to determine the testing coverage using CoverJS

Frameworks

- `vue-loader` Loads and compiles Vue Components
- `polymer-loader` Process HTML & CSS with preprocessor of choice and `require()` Web Components like first-class modules
- `angular2-template-loader` Loads and compiles Angular Components

For more third-party loaders, see the list from awesome-webpack.

webpack has a rich plugin interface. Most of the features within webpack itself use this plugin interface. This makes webpack **flexible**.

Name	Description
`AggressiveSplittingPlugin`	Splits the original chunks into smaller chunks
`BabelMinifyWebpackPlugin`	Minification with babel-minify
`BannerPlugin`	Add a banner to the top of each generated chunk
`CommonsChunkPlugin`	Extract common modules shared between chunks
`CompressionWebpackPlugin`	Prepare compressed versions of assets to serve them with Content-Encoding
`ContextReplacementPlugin`	Override the inferred context of a `require` expression
`CopyWebpackPlugin`	Copies individual files or entire directories to the build directory
`DefinePlugin`	Allow global constants configured at compile time
`DllPlugin`	Split bundles in order to drastically improve build time
`EnvironmentPlugin`	Shorthand for using the `DefinePlugin` on `process.env` keys
`ExtractTextWebpackPlugin`	Extract text (CSS) from your bundles into a separate file
`HotModuleReplacementPlugin`	Enable Hot Module Replacement (HMR)
`HtmlWebpackPlugin`	Easily create HTML files to serve your bundles
`I18nWebpackPlugin`	Add i18n support to your bundles
`IgnorePlugin`	Exclude certain modules from bundles
`LimitChunkCountPlugin`	Set min/max limits for chunking to better control chunking
`LoaderOptionsPlugin`	Used for migrating from webpack 1 to 2
`MinChunkSizePlugin`	Keep chunk size above the specified limit
`NoEmitOnErrorsPlugin`	Skip the emitting phase when there are compilation errors
`NormalModuleReplacementPlugin`	Replace resource(s) that matches a regexp
`NpmInstallWebpackPlugin`	Auto-install missing dependencies during development
`ProvidePlugin`	Use modules without having to use import/require
`SourceMapDevToolPlugin`	Enables a more fine grained control of source maps
`EvalSourceMapDevToolPlugin`	Enables a more fine grained control of eval source maps
`UglifyjsWebpackPlugin`	Enables control of the version of UglifyJS in your project
`ZopfliWebpackPlugin`	Prepare compressed versions of assets with node-zopfli

For more third-party plugins, see the list from awesome-webpack.

The `AggressiveSplittingPlugin` can split bundles into smaller chunks, splitting every chunk until it reaches the specified `maxSize` configured in `options`. It groups modules together by folder structure.

It records the split points in webpack records and tries to restore splitting in the same manner it started. This ensures that after changes to the application, the previous split points (and chunks) are reused as they are probably already in the client's cache. Therefore it's heavily recommended to use records.

Only chunks bigger than the specified `minSize` are stored in records. This ensures the chunks fill up as your application grows, instead of creating too many chunks for every change.

Chunks can be invalidated if a module changes. Modules from invalid chunks will go back into the module pool from which new chunks are created.

```
new webpack.optimize.AggressiveSplittingPlugin(options)
```

Options

```
{
  minSize: 30000, //Byte, split point. Default: 30720
  maxSize: 50000, //Byte, maxsize of per file. Default: 51200
  chunkOverhead: 0, //Default: 0
  entryChunkMultiplicator: 1, //Default: 1
}
```

Examples

http2-aggressive-splitting

Adds a banner to the top of each generated chunk.

```
new webpack.BannerPlugin(banner)
// or
new webpack.BannerPlugin(options)
```

Options

```
{
  banner: string, // the banner as string, it will be wrapped in a comment
  raw: boolean, // if true, banner will not be wrapped in a comment
  entryOnly: boolean, // if true, the banner will only be added to the entry chunks
  test: string | RegExp | Array,
  include: string | RegExp | Array,
  exclude: string | RegExp | Array,
}
```

Placeholders

Since webpack 2.5.0, placeholders are evaluated in the `banner` string:

```
new webpack.BannerPlugin({
  banner: "hash:[hash], chunkhash:[chunkhash], name:[name], filebase:[filebase], query:[query], file:[file]"
})
```

The `CommonsChunkPlugin` is an opt-in feature that creates a separate file (known as a chunk), consisting of common modules shared between multiple entry points.

W> The CommonsChunkPlugin has been removed in webpack v4 legato. To learn how chunks are treated in the latest version, check out the SplitChunksPlugin.

By separating common modules from bundles, the resulting chunked file can be loaded once initially, and stored in cache for later use. This results in pagespeed optimizations as the browser can quickly serve the shared code from cache, rather than being forced to load a larger bundle whenever a new page is visited.

```
new webpack.optimize.CommonsChunkPlugin(options)
```

Options

```
{
  name: string, // or
  names: string[],
  // The chunk name of the commons chunk. An existing chunk can be selected by passing a name of an existing ch
unk.
  // If an array of strings is passed this is equal to invoking the plugin multiple times for each chunk name.
  // If omitted and `options.async` or `options.children` is set all chunks are used, otherwise `options.filena
me`
  // is used as chunk name.
  // When using `options.async` to create common chunks from other async chunks you must specify an entry-point
  // chunk name here instead of omitting the `option.name`.

  filename: string,
  // The filename template for the commons chunk. Can contain the same placeholders as `output.filename`.
  // If omitted the original filename is not modified (usually `output.filename` or `output.chunkFilename`).
  // This option is not permitted if you're using `options.async` as well, see below for more details.

  minChunks: number|Infinity|function(module, count) -> boolean,
  // The minimum number of chunks which need to contain a module before it's moved into the commons chunk.
  // The number must be greater than or equal 2 and lower than or equal to the number of chunks.
  // Passing `Infinity` just creates the commons chunk, but moves no modules into it.
  // By providing a `function` you can add custom logic. (Defaults to the number of chunks)

  chunks: string[],
  // Select the source chunks by chunk names. The chunk must be a child of the commons chunk.
  // If omitted all entry chunks are selected.

  children: boolean,
  // If `true` all children of the commons chunk are selected

  deepChildren: boolean,
  // If `true` all descendants of the commons chunk are selected

  async: boolean|string,
  // If `true` a new async commons chunk is created as child of `options.name` and sibling of `options.chunks`.
  // It is loaded in parallel with `options.chunks`.
  // Instead of using `option.filename`, it is possible to change the name of the output file by providing
  // the desired string here instead of `true`.

  minSize: number,
  // Minimum size of all common module before a commons chunk is created.
}
```

T> The deprecated webpack 1 constructor `new webpack.optimize.CommonsChunkPlugin(options, filenameTemplate, selectedChunks, minChunks)` is no longer supported. Use a corresponding options object instead.

Examples

Commons chunk for entries

Generate an extra chunk, which contains common modules shared between entry points.

```
new webpack.optimize.CommonsChunkPlugin({
  name: "commons",
  // (the commons chunk name)

  filename: "commons.js",
  // (the filename of the commons chunk)

  // minChunks: 3,
  // (Modules must be shared between 3 entries)

  // chunks: ["pageA", "pageB"],
  // (Only use these entries)
})
```

You must load the generated chunk before the entry point:

```
<script src="commons.js" charset="utf-8"></script>
<script src="entry.bundle.js" charset="utf-8"></script>
```

Explicit vendor chunk

Split your code into vendor and application.

```
entry: {
  vendor: ["jquery", "other-lib"],
  app: "./entry"
},
plugins: [
  new webpack.optimize.CommonsChunkPlugin({
    name: "vendor",
    // filename: "vendor.js"
    // (Give the chunk a different name)

    minChunks: Infinity,
    // (with more entries, this ensures that no other module
    //  goes into the vendor chunk)
  })
]
```

```
<script src="vendor.js" charset="utf-8"></script>
<script src="app.js" charset="utf-8"></script>
```

T> In combination with long term caching you may need to use the `ChunkManifestWebpackPlugin` to avoid the vendor chunk changes. You should also use records to ensure stable module ids, e.g. using `NamedModulesPlugin` or `HashedModuleIdsPlugin`.

Move common modules into the parent chunk

With Code Splitting, multiple child chunks of an entry chunk can have common dependencies. To prevent duplication these can be moved into the parent. This reduces overall size, but does have a negative effect on the initial load time. If it is expected that users will need to download many sibling chunks, i.e. children of the entry chunk, then this should

improve load time overall.

```
new webpack.optimize.CommonsChunkPlugin({
  // names: ["app", "subPageA"]
  // (choose the chunks, or omit for all chunks)

  children: true,
  // (select all children of chosen chunks)

  // minChunks: 3,
  // (3 children must share the module before it's moved)
})
```

Extra async commons chunk

Similar to the above one, but instead of moving common modules into the parent (which increases initial load time) a new async-loaded additional commons chunk is used. This is automatically downloaded in parallel when the additional chunk is downloaded.

```
new webpack.optimize.CommonsChunkPlugin({
  name: "app",
  // or
  names: ["app", "subPageA"],
  // the name or list of names must match the name or names
  // of the entry points that create the async chunks

  children: true,
  // (use all children of the chunk)

  async: true,
  // (create an async commons chunk)

  minChunks: 3,
  // (3 children must share the module before it's separated)
})
```

Passing the `minChunks` property a function

You also have the ability to pass the `minChunks` property a function. This function is called by the `CommonsChunkPlugin` and calls the function with `module` and `count` arguments.

The `module` argument represents each module in the chunks you have provided via the `name` / `names` property. `module` has the shape of a NormalModule, which has two particularly useful properties for this use case:

- `module.context` : The directory that stores the file. For example: `'/my_project/node_modules/example-dependency'`
- `module.resource` : The name of the file being processed. For example: `'/my_project/node_modules/example-dependency/index.js'`

The `count` argument represents how many chunks the `module` is used in.

This option is useful when you want to have fine-grained control over how the CommonsChunk algorithm determines where modules should be moved to.

```
new webpack.optimize.CommonsChunkPlugin({
  name: "my-single-lib-chunk",
  filename: "my-single-lib-chunk.js",
  minChunks: function(module, count) {
    // If module has a path, and inside of the path exists the name "somelib",
    // and it is used in 3 separate chunks/entries, then break it out into
    // a separate chunk with chunk keyname "my-single-lib-chunk", and filename "my-single-lib-chunk.js"
    return module.resource && (/somelib/).test(module.resource) && count === 3;
```

```
  }
});
```

As seen above, this example allows you to move only one lib to a separate file if and only if all conditions are met inside the function.

This concept may be used to obtain implicit common vendor chunks:

```
new webpack.optimize.CommonsChunkPlugin({
  name: "vendor",
  minChunks: function (module) {
    // this assumes your vendor imports exist in the node_modules directory
    return module.context && module.context.includes("node_modules");
  }
})
```

In order to obtain a single CSS file containing your application and vendor CSS, use the following `minChunks` function together with `ExtractTextPlugin`:

```
new webpack.optimize.CommonsChunkPlugin({
  name: "vendor",
  minChunks: function (module) {
    // This prevents stylesheet resources with the .css or .scss extension
    // from being moved from their original chunk to the vendor chunk
    if(module.resource && (/^.*\.(css|scss)$/).test(module.resource)) {
      return false;
    }
    return module.context && module.context.includes("node_modules");
  }
})
```

Manifest file

To extract the webpack bootstrap logic into a separate file, use the `CommonsChunkPlugin` on a `name` which is not defined as `entry`. Commonly the name `manifest` is used. See the caching guide for details.

```
new webpack.optimize.CommonsChunkPlugin({
  name: "manifest",
  minChunks: Infinity
})
```

Combining implicit common vendor chunks and manifest file

Since the `vendor` and `manifest` chunk use a different definition for `minChunks`, you need to invoke the plugin twice:

```
[
  new webpack.optimize.CommonsChunkPlugin({
    name: "vendor",
    minChunks: function(module){
      return module.context && module.context.includes("node_modules");
    }
  }),
  new webpack.optimize.CommonsChunkPlugin({
    name: "manifest",
    minChunks: Infinity
  }),
```

```
]
```

More Examples

- Common and Vendor Chunks
- Multiple Common Chunks
- Multiple Entry Points with Commons Chunk

Context refers to a require with an expression such as `require('./locale/' + name + '.json')`. When encountering such an expression, webpack infers the directory (`'./locale/'`) and a regular expression (`/^.*\.json$/`). Since the `name` is not known at compile time, webpack includes every file as module in the bundle.

The `ContextReplacementPlugin` allows you to override the inferred information. There are various ways to configure the plugin:

Usage

```
new webpack.ContextReplacementPlugin(
  resourceRegExp: RegExp,
  newContentResource?: string,
  newContentRecursive?: boolean,
  newContentRegExp?: RegExp
)
```

If the resource (directory) matches `resourceRegExp`, the plugin replaces the default resource, recursive flag or generated regular expression with `newContentResource`, `newContentRecursive` or `newContextRegExp` respectively. If `newContentResource` is relative, it is resolved relative to the previous resource.

Here's a small example to restrict module usage:

```
new webpack.ContextReplacementPlugin(
  /moment[\/\\]locale$/,
  /de|fr|hu/
)
```

The `moment/locale` context is restricted to files matching `/de|fr|hu/`. Thus only those locales are included (see this issue for more information).

Content Callback

```
new webpack.ContextReplacementPlugin(
  resourceRegExp: RegExp,
  newContentCallback: (data) => void
)
```

The `newContentCallback` function is given a `data` object of the `ContextModuleFactory` and is expected to overwrite the `request` attribute of the supplied object.

Using this callback we can dynamically redirect requests to a new location:

```
new webpack.ContextReplacementPlugin(/^\.\/locale$/, (context) => {
  if ( !/\/moment\//.test(context.context) ) return;

  Object.assign(context, {
    regExp: /^\.\/\w+/,
    request: '../../locale' // resolved relatively
  });
}),
```

Other Options

The `newContentResource` and `newContentCreateContextMap` parameters are also available:

```
new webpack.ContextReplacementPlugin(
  resourceRegExp: RegExp,
  newContentResource: string,
  newContentCreateContextMap: object // mapping runtime-request (userRequest) to compile-time-request (request)
)
```

These two parameters can be used together to redirect requests in a more targeted way. The `newContentCreateContextMap` allows you to map runtime requests to compile requests in the form of an object:

```
new ContextReplacementPlugin(/selector/, './folder', {
  './request': './request',
  './other-request': './new-request'
})
```

The `DefinePlugin` allows you to create global constants which can be configured at **compile** time. This can be useful for allowing different behavior between development builds and release builds. If you perform logging in your development build but not in the release build you might use a global constant to determine whether logging takes place. That's where `DefinePlugin` shines, set it and forget it rules for development and release builds.

```
new webpack.DefinePlugin({
  // Definitions...
})
```

Usage

Each key passed into `DefinePlugin` is an identifier or multiple identifiers joined with `.`.

- If the value is a string it will be used as a code fragment.
- If the value isn't a string, it will be stringified (including functions).
- If the value is an object all keys are defined the same way.
- If you prefix `typeof` to the key, it's only defined for typeof calls.

The values will be inlined into the code allowing a minification pass to remove the redundant conditional.

```
new webpack.DefinePlugin({
  PRODUCTION: JSON.stringify(true),
  VERSION: JSON.stringify("5fa3b9"),
  BROWSER_SUPPORTS_HTML5: true,
  TWO: "1+1",
  "typeof window": JSON.stringify("object")
})
```

```
console.log("Running App version " + VERSION);
if(!BROWSER_SUPPORTS_HTML5) require("html5shiv");
```

T> Note that because the plugin does a direct text replacement, the value given to it must include **actual quotes** inside of the string itself. Typically, this is done either with either alternate quotes, such as `'"production"'`, or by using `JSON.stringify('production')`.

index.js

```
if (!PRODUCTION) {
  console.log('Debug info')
}

if (PRODUCTION) {
  console.log('Production log')
}
```

After passing through webpack with no minification results in:

```
if (!true) {
  console.log('Debug info')
}
if (true) {
  console.log('Production log')
}
```

and then after a minification pass results in:

```
console.log('Production log')
```

Feature Flags

Enable/disable features in production/development build using feature flags.

```
new webpack.DefinePlugin({
  'NICE_FEATURE': JSON.stringify(true),
  'EXPERIMENTAL_FEATURE': JSON.stringify(false)
})
```

W> When defining values for `process` prefer `'process.env.NODE_ENV': JSON.stringify('production')` over `process: { env: { NODE_ENV: JSON.stringify('production') } }`. Using the latter will overwrite the `process` object which can break compatibility with some modules that expect other values on the process object to be defined.

Service URLs

Use a different service URL in production/development builds:

```
new webpack.DefinePlugin({
  'SERVICE_URL': JSON.stringify("http://dev.example.com")
})
```

The `DllPlugin` and `DllReferencePlugin` provide means to split bundles in a way that can drastically improve build time performance.

`DllPlugin`

This plugin is used in a separate webpack config exclusively to create a dll-only-bundle. It creates a `manifest.json` file, which is used by the `DllReferencePlugin` to map dependencies.

- `context` (optional): context of requests in the manifest file (defaults to the webpack context.)
- `name` : name of the exposed dll function (TemplatePaths: `[hash]` & `[name]`)
- `path` : **absolute path** to the manifest json file (output)

```
new webpack.DllPlugin(options)
```

Creates a `manifest.json` which is written to the given `path` . It contains mappings from require and import requests, to module ids. It is used by the `DllReferencePlugin` .

Combine this plugin with `output.library` option to expose (aka, put into the global scope) the dll function.

`DllReferencePlugin`

This plugin is used in the primary webpack config, it references the dll-only-bundle(s) to require pre-built dependencies.

- `context` : (**absolute path**) context of requests in the manifest (or content property)
- `manifest` : an object containing `content` and `name` or a string to the absolute path of the JSON manifest to be loaded upon compilation
- `content` (optional): the mappings from request to module id (defaults to `manifest.content`)
- `name` (optional): the name where the dll is exposed (defaults to `manifest.name`) (see also `externals`)
- `scope` (optional): prefix which is used for accessing the content of the dll
- `sourceType` (optional): how the dll is exposed (libraryTarget)

```
new webpack.DllReferencePlugin(options)
```

References a dll manifest file to map dependency names to module ids, then requires them as needed using the internal `__webpack_require__` function.

W> Keep the `name` consistent with `output.library` .

Modes

This plugin can be used in two different modes, *scoped* and *mapped*.

Scoped Mode

The content of the dll is accessible under a module prefix. i.e. with `scope = "xyz"` a file `abc` in the dll can be access via `require("xyz/abc")` .

T> See an example use of scope

Mapped Mode

The content of the dll is mapped to the current directory. If a required file matches a file in the dll (after resolving), then the file from the dll is used instead.

Because this happens after resolving every file in the dll bundle, the same paths must be available for the consumer of the dll bundle. i.e. if the dll contains `lodash` and the file `abc`, `require("lodash")` and `require("./abc")` will be used from the dll, rather than building them into the main bundle.

Usage

W> `DllReferencePlugin` and `DllPlugin` are used in *separate* webpack configs.

webpack.vendor.config.js

```
new webpack.DllPlugin({
  context: __dirname,
  name: "[name]_[hash]",
  path: path.join(__dirname, "manifest.json"),
})
```

webpack.app.config.js

```
new webpack.DllReferencePlugin({
  context: __dirname,
  manifest: require("./manifest.json"),
  name: "./my-dll.js",
  scope: "xyz",
  sourceType: "commonjs2"
})
```

Examples

Vendor and User

Two separate example folders. Demonstrates scope and context.

T> Multiple `DllPlugins` and multiple `DllReferencePlugins`.

References

Source

- DllPlugin source
- DllReferencePlugin source
- DllEntryPlugin source
- DllModuleFactory source
- ManifestPlugin source

Tests

- DllPlugin creation test
- DllPlugin without scope test
- DllReferencePlugin use Dll test

The `EnvironmentPlugin` is shorthand for using the `DefinePlugin` on `process.env` keys.

Usage

The `EnvironmentPlugin` accepts either an array of keys or an object mapping its keys to their default values.

```
new webpack.EnvironmentPlugin(['NODE_ENV', 'DEBUG'])
```

This is equivalent to the following `DefinePlugin` application:

```
new webpack.DefinePlugin({
  'process.env.NODE_ENV': JSON.stringify(process.env.NODE_ENV),
  'process.env.DEBUG': JSON.stringify(process.env.DEBUG)
})
```

T> Not specifying the environment variable raises an " `EnvironmentPlugin` - `${key}` environment variable is undefined" error.

Usage with default values

Alternatively, the `EnvironmentPlugin` supports an object, which maps keys to their default values. The default value for a key is taken if the key is undefined in `process.env` .

```
new webpack.EnvironmentPlugin({
  NODE_ENV: 'development', // use 'development' unless process.env.NODE_ENV is defined
  DEBUG: false
})
```

W> Variables coming from `process.env` are always strings.

T> Unlike `DefinePlugin` , default values are applied to `JSON.stringify` by the `EnvironmentPlugin` .

T> To specify an unset default value, use `null` instead of `undefined` .

Example:

Let's investigate the result when running the previous `EnvironmentPlugin` configuration on a test file `entry.js` :

```
if (process.env.NODE_ENV === 'production') {
  console.log('Welcome to production');
}
if (process.env.DEBUG) {
  console.log('Debugging output');
}
```

When executing `NODE_ENV=production webpack` in the terminal to build, `entry.js` becomes this:

```
if ('production' === 'production') { // <-- 'production' from NODE_ENV is taken
  console.log('Welcome to production');
}
if (false) { // <-- default value is taken
  console.log('Debugging output');
}
```

Running `DEBUG=false webpack` yields:

```
if ('development' === 'production') { // <-- default value is taken
  console.log('Welcome to production');
}
if ('false') { // <-- 'false' from DEBUG is taken
  console.log('Debugging output');
}
```

`DotenvPlugin`

The third-party `DotenvPlugin` (`dotenv-webpack`) allows you to expose (a subset of) dotenv variables:

```
// .env
DB_HOST=127.0.0.1
DB_PASS=foobar
S3_API=mysecretkey
```

```
 new Dotenv({
  path: './.env', // Path to .env file (this is the default)
  safe: true // load .env.example (defaults to "false" which does not use dotenv-safe)
})
```

This plugin will cause hashes to be based on the relative path of the module, generating a four character string as the module id. Suggested for use in production.

```
new webpack.HashedModuleIdsPlugin({
  // Options...
})
```

Options

This plugin supports the following options:

- `hashFunction` : The hashing algorithm to use, defaults to `'md5'` . All functions from Node.JS' `crypto.createHash` are supported.
- `hashDigest` : The encoding to use when generating the hash, defaults to `'base64'` . All encodings from Node.JS' `hash.digest` are supported.
- `hashDigestLength` : The prefix length of the hash digest to use, defaults to `4` .

Usage

Here's an example of how this plugin might be used:

```
new webpack.HashedModuleIdsPlugin({
  hashFunction: 'sha256',
  hashDigest: 'hex',
  hashDigestLength: 20
})
```

Enables Hot Module Replacement, otherwise known as HMR.

W> HMR should **never** be used in production.

Basic Usage

Enabling HMR is easy and in most cases no options are necessary.

```
new webpack.HotModuleReplacementPlugin({
  // Options...
})
```

Options

The following options are accepted:

- `multiStep` (boolean): If `true`, the plugin will build in two steps -- first compiling the hot update chunks, and then the remaining normal assets.
- `fullBuildTimeout` (number): The delay between the two steps when `multiStep` is enabled.
- `requestTimeout` (number): The timeout used for manifest download (since webpack 3.0.0)

W> These options are experimental and may be deprecated. As mentioned above, they are typically not necessary and including a `new webpack.HotModuleReplacementPlugin()` is enough.

The `HtmlWebpackPlugin` simplifies creation of HTML files to serve your webpack bundles. This is especially useful for webpack bundles that include a hash in the filename which changes every compilation. You can either let the plugin generate an HTML file for you, supply your own template using lodash templates, or use your own loader.

Installation

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

Basic Usage

The plugin will generate an HTML5 file for you that includes all your webpack bundles in the body using `script` tags. Just add the plugin to your webpack config as follows:

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

var webpackConfig = {
  entry: 'index.js',
  output: {
    path: path.resolve(__dirname, './dist'),
    filename: 'index_bundle.js'
  },
  plugins: [new HtmlWebpackPlugin()]
};
```

This will generate a file `dist/index.html` containing the following:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>webpack App</title>
  </head>
  <body>
    <script src="index_bundle.js"></script>
  </body>
</html>
```

If you have multiple webpack entry points, they will all be included with `script` tags in the generated HTML.

If you have any CSS assets in webpack's output (for example, CSS extracted with the ExtractTextPlugin) then these will be included with `<link>` tags in the HTML head.

Configuration

For all configuration options, please see the plugin documentation.

Third party addons

The plugin supports addons. For a list see the documentation.

While writing your code, you may have already added many code split points to load stuff on demand. After compiling you might notice that some chunks are too small - creating larger HTTP overhead. Luckily, this plugin can post-process your chunks by merging them.

```
new webpack.optimize.LimitChunkCountPlugin({
  // Options...
})
```

Options

The following options are supported:

- `maxChunks` : Limit the maximum number of chunks using a value greater than or equal to `1`. Using `1` will prevent any additional chunks from being added as the entry/main chunk is also included in the count.
- `minChunkSize` : Set a minimum chunk size.

While merging chunks, webpack will try to identify those that have duplicate modules and merge them first. Nothing will be merged into the entry chunk, so as not to impact initial page loading time. Here's a small example:

```
new webpack.optimize.LimitChunkCountPlugin({
  maxChunks: 5, // Must be greater than or equal to one
  minChunkSize: 1000
})
```

CLI

This plugin and it's options can also be invoked via the CLI:

```
--optimize-max-chunks 15
```

or

```
--optimize-min-chunk-size 10000
```

The `LoaderOptionsPlugin` is unlike other plugins in that it is built for migration from webpack 1 to 2. In webpack 2, the schema for a `webpack.config.js` became stricter; no longer open for extension by other loaders and plugins. The intention is that you pass `options` directly to loaders and plugins (i.e. `options` are **not** global or shared).

However, until a loader has been updated to depend upon options being passed directly to them, the `LoaderOptionsPlugin` exists to bridge the gap. You can configure global loader options with this plugin and all loaders will receive these options.

```
new webpack.LoaderOptionsPlugin({
  // Options...
})
```

W> This plugin will be removed in the future as it only exists for migration.

Options

This plugin supports the following options:

- `options.debug` (`boolean`): Whether loaders should be in `debug` mode or not. `debug` will be removed as of webpack 3.
- `options.minimize` (`boolean`): Where loaders can be switched to minimize mode.
- `options.options` (`object`): A configuration object that can be used to configure older loaders - this will take the same schema a `webpack.config.js` .
- `options.options.context` (`string`): The context that can be used to configure older loaders.
- any other options allowed in a `webpack.config.js` ....

Usage

Here's an example of how this plugin might be used:

```
new webpack.LoaderOptionsPlugin({
  minimize: true,
  debug: false,
  options: {
    context: __dirname
  }
})
```

Keep chunk size above the specified limit by merging chunks that are smaller than the `minChunkSize`.

```
new webpack.optimize.MinChunkSizePlugin({
  minChunkSize: 10000 // Minimum number of characters
})
```

This plugin will cause the relative path of the module to be displayed when HMR is enabled. Suggested for use in development.

```
new webpack.NamedModulesPlugin()
```

Use the `NoEmitOnErrorsPlugin` to skip the emitting phase whenever there are errors while compiling. This ensures that no assets are emitted that include errors. The `emitted` flag in the stats is `false` for all assets.

```
new webpack.NoEmitOnErrorsPlugin()
```

T> This supersedes the (now deprecated) webpack 1 plugin `NoErrorsPlugin`.

W> If you are using the CLI, the webpack process will not exit with an error code by enabling this plugin. If you want webpack to "fail" when using the CLI, please check out the `bail` option.

Automatically load modules instead of having to `import` or `require` them everywhere.

```
new webpack.ProvidePlugin({
  identifier: 'module1',
  // ...
})
```

or

```
new webpack.ProvidePlugin({
  identifier: ['module1', 'property1'],
  // ...
})
```

Whenever the `identifier` is encountered as free variable in a module, the `module` is loaded automatically and the `identifier` is filled with the exports of the loaded `module` (of `property` in order to support named exports).

W> For importing the default export of an ES2015 module, you have to specify the default property of module.

Usage: jQuery

To automatically load `jquery` we can simply point both variables it exposes to the corresponding node module:

```
new webpack.ProvidePlugin({
  $: 'jquery',
  jQuery: 'jquery'
})
```

Then in any of our source code:

```
// in a module
$('#item'); // <= just works
jQuery('#item'); // <= just works
// $ is automatically set to the exports of module "jquery"
```

Usage: jQuery with Angular 1

Angular looks for `window.jQuery` in order to determine whether jQuery is present, see the source code.

```
new webpack.ProvidePlugin({
  'window.jQuery': 'jquery'
})
```

Usage: Lodash Map

```
new webpack.ProvidePlugin({
  _map: ['lodash', 'map']
})
```

Usage: Vue.js

```
new webpack.ProvidePlugin({
  Vue: ['vue/dist/vue.esm.js', 'default']
})
```

This plugin enables more fine grained control of source map generation. It is an alternative to the `devtool` configuration option.

```
new webpack.SourceMapDevToolPlugin(options)
```

Options

The following options are supported:

- `test` (`string|regex|array`): Include source maps for modules based on their extension (defaults to `.js` and `.css`).
- `include` (`string|regex|array`): Include source maps for module paths that match the given value.
- `exclude` (`string|regex|array`): Exclude modules that match the given value from source map generation.
- `filename` (`string`): Defines the output filename of the SourceMap (will be inlined if no value is provided).
- `append` (`string`): Appends the given value to the original asset. Usually the `#sourceMappingURL` comment. `[url]` is replaced with a URL to the source map file. `false` disables the appending.
- `moduleFilenameTemplate` (`string`): See `output.devtoolModuleFilenameTemplate` .
- `fallbackModuleFilenameTemplate` (`string`): See link above.
- `module` (`boolean`): Indicates whether loaders should generate source maps (defaults to `true`).
- `columns` (`boolean`): Indicates whether column mappings should be used (defaults to `true`).
- `lineToLine` (`object`): Simplify and speed up source mapping by using line to line source mappings for matched modules.
- `noSources` (`boolean`): Prevents the source file content from being included in the source map (defaults to `false`).
- `publicPath` (`string`): Emits absolute URLs with public path prefix, e.g. `https://example.com/project/` .
- `fileContext` (`string`): Makes the `[file]` argument relative to this directory.

The `lineToLine` object allows for the same `test` , `include` , and `exclude` options described above.

The `fileContext` option is useful when you want to store source maps in an upper level directory to avoid `../../` appearing in the absolute `[url]` .

T> Setting `module` and/or `columns` to `false` will yield less accurate source maps but will also improve compilation performance significantly.

W> Remember that when using the `UglifyJSPlugin` , you must utilize the `sourceMap` option.

Examples

The following examples demonstrate some common use cases for this plugin.

Exclude Vendor Maps

The following code would exclude source maps for any modules in the `vendor.js` bundle:

```
new webpack.SourceMapDevToolPlugin({
  filename: '[name].js.map',
  exclude: ['vendor.js']
})
```

Host Source Maps Externally

Set a URL for source maps. Useful for hosting them on a host that requires authorization.

```
new webpack.SourceMapDevToolPlugin({
  append: "\n//# sourceMappingURL=http://example.com/sourcemap/[url]",
  filename: '[name].map'
})
```

And for cases when source maps are stored in the upper level directory:

```
project
|- dist
  |- public
    |- bundle-[hash].js
  |- sourcemaps
    |- bundle-[hash].js.map
```

With next config:

```
new webpack.SourceMapDevToolPlugin({
  filename: "sourcemaps/[file].map",
  publicPath: "https://example.com/project/",
  fileContext: "public"
})
```

Will produce the following URL:

```
https://example.com/project/sourcemaps/bundle-[hash].js.map`
```

Originally, chunks (and modules imported inside them) were connected by a parent-child relationship in the internal webpack graph. The `CommonsChunkPlugin` was used to avoid duplicated dependencies across them, but further optimizations where not possible

Since version 4 the `CommonsChunkPlugin` was removed in favor of `optimization.splitChunks` and `optimization.runtimeChunk` options. Here is how the new flow works.

Defaults

Out of the box `SplitChunksPlugin` should work great for most users.

By default it only affects on-demand chunks because changing initial chunks would affect the script tags the HTML file should include to run the project.

webpack will automatically split chunks based on these conditions:

- New chunk can be shared OR modules are from the `node_modules` folder
- New chunk would be bigger than 30kb (before min+gz)
- Maximum number of parallel requests when loading chunks on demand would be lower or equal to 5
- Maximum number of parallel requests at initial page load would be lower or equal to 3

When trying to fulfill the last two conditions, bigger chunks are preferred.

Let's take a look at some examples.

Defaults: Example 1

```
// index.js

// dynamically import a.js
import("./a");
```

```
// a.js
import "react";

// ...
```

Result: A separate chunk would be created containing `react` . At the import call this chunk is loaded in parallel to the original chunk containing `./a` .

Why:

- Condition 1: The chunk contains modules from `node_modules`
- Condition 2: `react` is bigger than 30kb
- Condition 3: Number of parallel requests at the import call is 2
- Condition 4: Doesn't affect request at initial page load

What's the reasoning behind this? `react` probably won't change as often as your application code. By moving it into a separate chunk this chunk can be cached separately from your app code (assuming you are using chunkhash, records, Cache-Control or other long term cache approach).

Defaults: Example 2

```
// entry.js
```

```
// dynamically import a.js and b.js
import("./a");
import("./b");
```

```
// a.js
import "./helpers"; // helpers is 40kb in size

// ...
```

```
// b.js
import "./helpers";
import "./more-helpers"; // more-helpers is also 40kb in size

// ...
```

Result: A separate chunk would be created containing `./helpers` and all dependencies of it. At the import calls this chunk is loaded in parallel to the original chunks.

Why:

- Condition 1: The chunk is shared between both import calls
- Condition 2: `helpers` is bigger than 30kb
- Condition 3: Number of parallel requests at the import calls is 2
- Condition 4: Doesn't affect request at initial page load

Putting the content of `helpers` into each chunk will result into its code being downloaded twice. By using a separate chunk this will only happen once. We pay the cost of an additional request, which could be considered a tradeoff. That's why there is a minimum size of 30kb.

Configuration

For developers that want to have more control over this functionality, webpack provides a set of options to better fit your needs.

If you are manually changing the split configuration, measure the impact of the changes to see and make sure there's a real benefit.

W> Default configuration was chosen to fit web performance best practices but the optimum strategy for your project might defer depending on the nature of it.

Configuring cache groups

The defaults assign all modules from `node_modules` to a cache group called `vendors` and all modules duplicated in at least 2 chunks to a cache group `default`.

A module can be assigned to multiple cache groups. The optimization then prefers the cache group with the higher `priority` (`priority` option) or that one that forms bigger chunks.

Conditions

Modules from the same chunks and cache group will form a new chunk when all conditions are fulfilled.

There are 4 options to configure the conditions:

- `minSize` (default: 30000) Minimum size for a chunk.

- `minChunks` (default: 1) Minimum number of chunks that share a module before splitting
- `maxInitialRequests` (default 3) Maximum number of parallel requests at an entrypoint
- `maxAsyncRequests` (default 5) Maximum number of parallel requests at on-demand loading

Naming

To control the chunk name of the split chunk the `name` option can be used.

W> When assigning equal names to different split chunks, all vendor modules are placed into a single shared chunk, though it's not recommend since it can result in more code downloaded.

The magic value `true` automatically chooses a name based on chunks and cache group key, otherwise a string or function can be passed.

When the name matches an entry point name, the entry point is removed.

`optimization.splitChunks.automaticNameDelimiter`

By default webpack will generate names using origin and name of the chunk, like `vendors~main.js`.

If your project has a conflict with the `~` character, it can be changed by setting this option to any other value that works for your project: `automaticNameDelimiter: "-"`.

Then the resulting names will look like `vendors-main.js`.

Select modules

The `test` option controls which modules are selected by this cache group. Omitting it selects all modules. It can be a RegExp, string or function.

It can match the absolute module resource path or chunk names. When a chunk name is matched, all modules in this chunk are selected.

Select chunks

With the `chunks` option the selected chunks can be configured.

There are 3 values possible `"initial"`, `"async"` and `"all"`. When configured the optimization only selects initial chunks, on-demand chunks or all chunks.

The option `reuseExistingChunk` allows to reuse existing chunks instead of creating a new one when modules match exactly.

This can be controlled per cache group.

`optimization.splitChunks.chunks: all`

As it was mentioned before this plugin will affect dynamic imported modules. Setting the `optimization.splitChunks.chunks` option to `"all"` initial chunks will get affected by it (even the ones not imported dynamically). This way chunks can even be shared between entry points and on-demand loading.

This is the recommended configuration.

T> You can combine this configuration with the HtmlWebpackPlugin, it will inject all the generated vendor chunks for you.

`optimization.splitChunks`

This configuration object represents the default behavior of the `SplitChunksPlugin`.

```
splitChunks: {
    chunks: "async",
    minSize: 30000,
    minChunks: 1,
    maxAsyncRequests: 5,
    maxInitialRequests: 3,
    automaticNameDelimiter: '~',
    name: true,
    cacheGroups: {
        vendors: {
            test: /[\\/]node_modules[\\/]/,
            priority: -10
        },
    default: {
            minChunks: 2,
            priority: -20,
            reuseExistingChunk: true
        }
    }
}
```

By default cache groups inherit options from `splitChunks.*`, but `test`, `priority` and `reuseExistingChunk` can only be configured on cache group level.

`cacheGroups` is an object where keys are the cache group names. All options from the ones listed above are possible: `chunks`, `minSize`, `minChunks`, `maxAsyncRequests`, `maxInitialRequests`, `name`.

You can set `optimization.splitChunks.cacheGroups.default` to `false` to disable the default cache group, same for `vendors` cache group.

The priority of the default groups are negative to allow any custom cache group to take higher priority (the default value is `0`).

Here are some examples and their effect:

Split Chunks: Example 1

Create a `commons` chunk, which includes all code shared between entry points.

```
splitChunks: {
    cacheGroups: {
        commons: {
            name: "commons",
            chunks: "initial",
            minChunks: 2
        }
    }
}
```

W> This configuration can enlarge your initial bundles, it is recommended to use dynamic imports when a module is not immediately needed.

Split Chunks: Example 2

Create a `vendors` chunk, which includes all code from `node_modules` in the whole application.

```
splitChunks: {
    cacheGroups: {
        commons: {
            test: /[\\/]node_modules[\\/]/,
            name: "vendors",
            chunks: "all"
        }
    }
}
```

W> This might result in a large chunk containing all external packages. It is recommended to only include your core frameworks and utilities and dynamically load the rest of the dependencies.

`optimization.runtimeChunk`

Setting `optimization.runtimeChunk` to `true` adds an additonal chunk to each entrypoint containing only the runtime.

The value `single` instead creates a runtime file to be shared for all generated chunks.

Ignore the specified files, i.e. those matching the provided paths or regular expressions, while in watch mode.

```
new webpack.WatchIgnorePlugin(paths)
```

Options

- `paths` (array): A list of RegExps or absolute paths to directories or files that should be ignored

Prevent generation of modules for `import` or `require` calls matching the following regular expressions:

- `requestRegExp` A RegExp to test the request against.
- `contextRegExp` (optional) A RegExp to test the context (directory) against.

```
new webpack.IgnorePlugin(requestRegExp, [contextRegExp])
```

The following examples demonstrate a few ways this plugin can be used.

Ignore Moment Locales

As of moment 2.18, all locales are bundled together with the core library (see this GitHub issue). You can use the `IgnorePlugin` to stop any locale being bundled with moment:

```
new webpack.IgnorePlugin(/^\.\/locale$/, /moment$/)
```