Printable Configuration

Out of the box, webpack won't require you to use a configuration file. However, it will assume the entry point of your project is `src/index` and will output the result in `dist/main.js` minified and optimized for production.

Usually your projects will need to extend this functionality, for this you can create a `webpack.config.js` file in the root folder and webpack will automatically use it.

All the available configuration options are specified below.

> *New to webpack? Check out our guide to some of webpack's core concepts to get started!*

Use different config file

If for some reason you want to use different config file depending on certain situations you can change this via command line by using the `--config` flag.

package.json

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

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.

> *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.

webpack.config.js

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

module.exports = {
  <mode "/configuration/mode">
    <default>
      mode: "production", // "production" | "development" | "none"
    </default>
    mode: "production", // enable many optimizations for production builds
    mode: "development", // enabled useful tools for development
    mode: "none", // no defaults
  </mode>
  // Chosen mode tells webpack to use its built-in optimizations accordingly.
  <entry "/configuration/entry-context/#entry">
    <default>
      entry: "./app/entry", // string | object | array
    </default>
    entry: ["./app/entry1", "./app/entry2"],
    entry: {
      a: "./app/entry-a",
      b: ["./app/entry-b1", "./app/entry-b2"]
    },
  </entry>
  // defaults to ./src
  // Here the application starts executing
  // and webpack starts bundling
  <link "/configuration/output">
    <default>
      output: {
    </default>
  </link>
    // options related to how webpack emits results
    path: path.resolve(__dirname, "dist"), // string
    // the target directory for all output files
    // must be an absolute path (use the Node.js path module)
    <filename "/configuration/output/#outputfilename">
      <default>
        filename: "bundle.js", // string
      </default>
      filename: "[name].js", // for multiple entry points
      filename: "[chunkhash].js", // for long term caching
    </filename>
    // the filename template for entry chunks
    <publicPath "/configuration/output/#outputpublicpath">
      <default>
        publicPath: "/assets/", // string
      </default>
```

```
  publicPath: "",
  publicPath: "https://cdn.example.com/",
</publicPath>
// the url to the output directory resolved relative to the HTML page
library: "MyLibrary", // string,
// the name of the exported library
<libraryTarget "/configuration/output/#outputlibrarytarget">
  <default>
    libraryTarget: "umd", // universal module definition
  </default>
  libraryTarget: "umd2", // universal module definition
  libraryTarget: "commonjs2", // exported with module.exports
  libraryTarget: "commonjs", // exported as properties to exports
  libraryTarget: "amd", // defined with AMD defined method
  libraryTarget: "this", // property set on this
  libraryTarget: "var", // variable defined in root scope
  libraryTarget: "assign", // blind assignment
  libraryTarget: "window", // property set to window object
  libraryTarget: "global", // property set to global object
  libraryTarget: "jsonp", // jsonp wrapper
</libraryTarget>
// the type of the exported library
<advancedOutput "#">
  <default>
    /* Advanced output configuration (click to show) */
  </default>
  pathinfo: true, // boolean
  // include useful path info about modules, exports, requests, etc. into the generated cod
  chunkFilename: "[id].js",
  chunkFilename: "[chunkhash].js", // for long term caching
  // the filename template for additional chunks
  jsonpFunction: "myWebpackJsonp", // string
  // name of the JSONP function used to load chunks
  sourceMapFilename: "[file].map", // string
  sourceMapFilename: "sourcemaps/[file].map", // string
  // the filename template of the source map location
  devtoolModuleFilenameTemplate: "webpack:///[resource-path]", // string
  // the name template for modules in a devtool
  devtoolFallbackModuleFilenameTemplate: "webpack:///[resource-path]?[hash]", // string
  // the name template for modules in a devtool (used for conflicts)
  umdNamedDefine: true, // boolean
  // use a named AMD module in UMD library
  crossOriginLoading: "use-credentials", // enum
  crossOriginLoading: "anonymous",
  crossOriginLoading: false,
  // specifies how cross origin request are issued by the runtime
</advancedOutput>
<expert "#">
  <default>
    /* Expert output configuration (on own risk) */
  </default>
```

```
    devtoolLineToLine: {
      test: /\\.jsx$/
    },
    // use a simple 1:1 mapped SourceMaps for these modules (faster)
    hotUpdateMainFilename: "[hash].hot-update.json", // string
    // filename template for HMR manifest
    hotUpdateChunkFilename: "[id].[hash].hot-update.js", // string
    // filename template for HMR chunks
    sourcePrefix: "\\t", // string
    // prefix module sources in bundle for better readablitity
  </expert>
},
module: {
  // configuration regarding modules
  rules: [
    // rules for modules (configure loaders, parser options, etc.)
    {
      test: /\\.jsx?$/,
      include: [
        path.resolve(__dirname, "app")
      ],
      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: { test, include, exclude },
      // conditions for the issuer (the origin of the import)
      enforce: "pre",
      enforce: "post",
      // flags to apply these rules, even if they are overridden (advanced option)
      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
      options: {
        presets: ["es2015"]
      },
      // options for the loader
    },
    {
      test: /\\.html$/,
      use: [
        // apply multiple loaders and options
        "htmllint-loader",
        {
```

```
          loader: "html-loader",
          options: {
            /* ... */
          }
        }
      ]
    },
    { oneOf: [ /* rules */ ] },
    // only use one of these nested rules
    { rules: [ /* rules */ ] },
    // use all of these nested rules (combine with conditions to be useful)
    { resource: { and: [ /* conditions */ ] } },
    // matches only if all conditions are matched
    { resource: { or: [ /* conditions */ ] } },
    { resource: [ /* conditions */ ] },
    // matches if any condition is matched (default for arrays)
    { resource: { not: /* condition */ } }
    // matches if the condition is not matched
  ],
  <advancedModule "#">
    <default>
      /* Advanced module configuration (click to show) */
    </default>
    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
  </advancedModule>
},
resolve: {
  // options for resolving module requests
  // (does not apply to resolving to loaders)
  modules: [
    "node_modules",
    path.resolve(__dirname, "app")
  ],
  // directories where to look for modules
  extensions: [".js", ".json", ".jsx", ".css"],
  // extensions that are used
```

```
alias: {
  // a list of module name aliases
  "module": "new-module",
```

Want to rapidly generate webpack configuration file for your project requirements with few clicks away.

Generate Custom Webpack Configuration is an interactive portal you can play around by selecting custom webpack configuration options tailored for your frontend project. It automatically generates a minimal webpack configuration based on your selection of loaders/plugins, etc.

Or use webpack-cli's `init` command that will ask you a couple of questions before creating a configuration file.

```
npx webpack-cli init
```

You might be prompted to install `@webpack-cli/init` if it is not yet installed in the project or globally.

After running `npx webpack-cli init` you might get more packages installed to your project depending on the choices you've made during configuration generation.

```
npx webpack-cli init

ℹ INFO For more information and a detailed description of each question, have a look at https://g
  INFO Alternatively, run `webpack(-cli) --help` for usage info.

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

+ babel-plugin-syntax-dynamic-import@6.18.0
+ uglifyjs-webpack-plugin@2.0.1
+ webpack-cli@3.2.3
+ @babel/core@7.2.2
+ babel-loader@8.0.4
+ @babel/preset-env@7.1.0
+ webpack@4.29.3
added 124 packages from 39 contributors, updated 4 packages and audited 25221 packages in 7.463s
found 0 vulnerabilities


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

createapp.dev - create a webpack config in your browser is an online tool for creating custom webpack configuration. It allows you to select various features that will be combined and added to resulting configuration file. Also, it generates an example project based on provided webpack configuration that you can review in your browser and download.

Configuration Languages

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, i.e., TypeScript and the relevant type definitions from the DefinitelyTyped project:

```
npm install --save-dev typescript ts-node @types/node @types/webpack
# and, if using webpack-dev-server
npm install --save-dev @types/webpack-dev-server
```

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",
    "esModuleInterop": true
  }
}
```

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

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

package.json

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

> *We had been getting reports that `TS_NODE_PROJECT` might not work with `"TS_NODE_PROJECT" unrecognized command` error. Therefore running it with `cross-env` seems to fix the issue, for more info see this issue.*

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 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>
      <CustomPlugin foo="bar" />
    </plugins>
  </webpack>
);
```

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.

Configuration Types

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 TerserPlugin({
         terserOptions: {
+          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.

> *It is possible to export multiple promises by wrapping them into `Promise.all([/* Your promises */])`.*

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

> *Returning a `Promise` only works when using webpack via CLI. `webpack()` expects an object.*

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

If you pass a name to `--config-name` flag, webpack will only build that specific configuration.

Entry and Context

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.

```
module.exports = {
  //...
  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 where to start the application bundling process. If an array is passed then all items will be processed.

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.

```
module.exports = {
  //...
  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 entry point for the chunk.

Dynamic entry

If a function is passed then it will be invoked on every make event.

> *Note that the make event triggers when webpack starts and for every invalidation when watching for file changes.*

```
module.exports = {
  //...
```

or

```
module.exports = {
  //...
  entry: () => new Promise((resolve) => resolve(['./demo', './demo2']))
};
```

For example: you can use dynamic entries to get the actual entries from an external source (remote server, file system content or database):

webpack.config.js

```
module.exports = {
  entry() {
    return fetchPathsFromSomeExternalSource(); // returns a promise that will be resolved with some
  }
};
```

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

Mode

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

```
string = 'production': 'none' | 'development' | 'production'
```

Usage

Just provide the `mode` option in the config:

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

or pass it as a CLI argument:

```
webpack --mode=development
```

The following string values are supported:

Option	`development`
Description	Sets `process.env.NODE_ENV` on `DefinePlugin` to value `development`. Enables `NamedChunksPlugin` and `NamedModulesPlugin`.
Option	`production`
Description	Sets `process.env.NODE_ENV` on `DefinePlugin` to value `production`. Enables `FlagDependencyUsagePlugin`, `FlagIncludedChunksPlugin`, `ModuleConcatenationPlugin`, `NoEmitOnErrorsPlugin`, `OccurrenceOrderPlugin`, `SideEffectsFlagPlugin` and `TerserPlugin`.
Option	`none`
Description	Opts out of any default optimization options

If not set, webpack sets `production` as the default value for `mode`.

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

Mode: development

```
// webpack.development.config.js
module.exports = {
+ mode: 'development'
- devtool: 'eval',
- cache: true,
- performance: {
-   hints: false
- },
- output: {
-   pathinfo: true
- },
- optimization: {
-   namedModules: true,
-   namedChunks: true,
-   nodeEnv: 'development',
-   flagIncludedChunks: false,
-   occurrenceOrder: false,
```

```
-   sideEffects: false,
-   usedExports: false,
-   concatenateModules: false,
-   splitChunks: {
-     hidePathInfo: false,
```

Mode: production

```
// webpack.production.config.js
module.exports = {
+  mode: 'production',
- performance: {
-   hints: 'warning'
- },
- output: {
-   pathinfo: false
- },
- optimization: {
-   namedModules: false,
-   namedChunks: false,
-   nodeEnv: 'production',
-   flagIncludedChunks: true,
-   occurrenceOrder: true,
-   sideEffects: true,
-   usedExports: true,
-   concatenateModules: true,
-   splitChunks: {
-     hidePathInfo: true,
-     minSize: 30000,
-     maxAsyncRequests: 5,
-     maxInitialRequests: 3,
-   },
-   noEmitOnErrors: true,
-   checkWasmTypes: true,
```

Mode: none

```
// webpack.custom.config.js
module.exports = {
+ mode: 'none',
- performance: {
-  hints: false
- },
- optimization: {
-   flagIncludedChunks: false,
-   occurrenceOrder: false,
-   sideEffects: false,
-   usedExports: false,
-   concatenateModules: false,
-   splitChunks: {
-     hidePathInfo: false,
-     minSize: 10000,
-     maxAsyncRequests: Infinity,
-     maxInitialRequests: Infinity,
-   },
-   noEmitOnErrors: false,
-   checkWasmTypes: false,
-   minimize: false,
- },
- plugins: []
}
```

If you want to change the behavior according to the **mode** variable inside the *webpack.config.js*, you have to export a function instead of an object:

```
var config = {
  entry: './app.js'
  //...
};

module.exports = (env, argv) => {

  if (argv.mode === 'development') {
```

```
    config.devtool = 'source-map';
  }
```

Output

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:

webpack.config.js

```
module.exports = {
  //...
  output: {
    library: 'someLibName',
    libraryTarget: 'umd',
    filename: 'someLibName.js',
    auxiliaryComment: 'Test Comment'
  }
};
```

which will yield the following:

webpack.config.js

```
(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:

webpack.config.js

```
module.exports = {
  //...
  output: {
    //...
    auxiliaryComment: {
      root: 'Root Comment',
      commonjs: 'CommonJS Comment',
      commonjs2: 'CommonJS2 Comment',
      amd: 'AMD Comment'
    }
  }
};
```

output.chunkFilename

`string = '[id].js'`

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).

webpack.config.js

```
module.exports = {
  //...
  output: {
    //...
    chunkFilename: '[id].js'
  }
};
```

output.chunkLoadTimeout

`number = 120000`

Number of milliseconds before chunk request expires. This option is supported since webpack 2.6.0.

webpack.config.js

```
module.exports = {
  //...
  output: {
    //...
    chunkLoadTimeout: 30000
  }
};
```

output.crossOriginLoading

`boolean = false  string: 'anonymous' | 'use-credentials'`

Tells webpack to enable cross-origin loading of chunks. Only takes effect when `target` is set to `'web'`, which uses JSONP for loading on-demand chunks, by adding script tags.

- `'anonymous'` - Enable cross-origin loading **without credentials**
- `'use-credentials'` - Enable cross-origin loading **with credentials**

output.jsonpScriptType

`string = 'text/javascript': 'module' | 'text/javascript'`

Allows customization of `type` attribute of `script` tags that webpack injects into the DOM to download async chunks.

- `'text/javascript'` : Default `type` in HTML5 and required for some browsers in HTML4.
- `'module'` : Causes the code to be treated as a JavaScript module.

output.devtoolFallbackModuleFilenameTemplate

`string  function (info)`

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

See `output.devtoolModuleFilenameTemplate` .

output.devtoolLineToLine

`boolean = false  object: { test string | RegExp, include string | RegExp, exclude string | RegExp}`

> *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`). Use `object` for granular control, e.g. to enable this feature for all javascript files within a certain directory:

webpack.config.js

```
module.exports = {
  //...
  output: {
    devtoolLineToLine: { test: /\.js$/, include: 'src/utilities' }
  }
};
```

output.devtoolModuleFilenameTemplate

`string = 'webpack://[namespace]/[resource-path]?[loaders]'  function (info) => string`

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'`.

webpack.config.js

```
module.exports = {
  //...
  output: {
    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
Template Description	[all-loaders] Automatic and explicit loaders and params up to the name of the first loader
Template Description	[hash] The hash of the module identifier
Template Description	[id] The module identifier
Template Description	[loaders] Explicit loaders and params up to the name of the first loader
Template Description	[resource] The path used to resolve the file and any query params used on the first loader
Template Description	[resource-path] The path used to resolve the file without any query params
Template	[namespace]

Description	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:

```
module.exports = {
  //...
  output: {
    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 (chunkData) => string`

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.

webpack.config.js

```
module.exports = {
  //...
  output: {
    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:

webpack.config.js

```
module.exports = {
  //...
  output: {
    filename: '[name].bundle.js'
  }
};
```

Using internal chunk id:

webpack.config.js

```
module.exports = {
  //...
  output: {
    filename: '[id].bundle.js'
  }
};
```

Using the unique hash generated for every build:

webpack.config.js

```
module.exports = {
  //...
  output: {
    filename: '[name].[hash].bundle.js'
  }
};
```

Using hashes based on each chunks' content:

webpack.config.js

```
module.exports = {
  //...
  output: {
    filename: '[chunkhash].bundle.js'
  }
};
```

Using hashes generated for extracted content:

webpack.config.js

```
module.exports = {
  //...
  output: {
    filename: '[contenthash].bundle.css'
  }
};
```

Using function to return the filename:

webpack.config.js

```
module.exports = {
  //...
  output: {
    filename: (chunkData) => {
      return chunkData.chunk.name === 'main' ? '[name].js': '[name]/[name].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	[hash]
Description	The hash of the module identifier

Template	[contenthash]
Description	the hash of the content of a file, which is different for each asset
Template	[chunkhash]
Description	The hash of the chunk content
Template	[name]
Description	The module name
Template	[id]
Description	The module identifier
Template	[query]
Description	The module query, i.e., the string following `?` in the filename
Template	[function]
Description	The function, which can return filename [string]

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

It is possible to filter out placeholder replacement when you want to use one of the placeholders in the actual file name. For example, to output a file `[name].js`, you have to escape the `[name]` placeholder by adding backslashes between the brackets. So that `[\name\]` generates `[name]` instead of getting replaced with the `name` of the asset.

Example: `[\id\]` generates `[id]` instead of getting replaced with the `id`.

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

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

output.globalObject

`string = 'window'`

When targeting a library, especially the `libraryTarget` is `'umd'`, this option indicates what global object will be used to mount the library. To make UMD build available on both browsers and Node.js, set `output.globalObject` option to `'this'`.

For example:

webpack.config.js

```
module.exports = {
  // ...
  output: {
    library: 'myLib',
    libraryTarget: 'umd',
    filename: 'myLib.js',
    globalObject: 'this'
  }
};
```

output.hashDigest

`string = 'hex'`

The encoding to use when generating the hash. All encodings from Node.JS' `hash.digest` are supported. Using `'base64'` for filenames might be problematic since it has the character `/` in its alphabet. Likewise `'latin1'` could contain any character.

output.hashDigestLength

`number = 20`

The prefix length of the hash digest to use.

output.hashFunction

`string = 'md4'  function`

The hashing algorithm to use. 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.

```
module.exports = {
  //...
  output: {
    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 = '[id].[hash].hot-update.js'  function (chunkData) => string`

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:

webpack.config.js

```
module.exports = {
  //...
  output: {
    hotUpdateChunkFilename: (chunkData) => {
      return `${chunkData.chunk.name === 'main' ? '' : '[name]/'}[id].[hash].hot-update.js`;
    }
  }
};
```

Typically you don't need to change `output.hotUpdateChunkFilename`.

output.hotUpdateFunction

`string`

Only used when `target` is set to `'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 = '[hash].hot-update.json'` `function`

Customize the main hot update filename. `[hash]` is the only available placeholder.

> *Typically you don't need to change `output.hotUpdateMainFilename`.*

output.jsonpFunction

`string = 'webpackJsonp'`

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

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

If using the `output.library` option, the library name is automatically concatenated with `output.jsonpFunction`'s value.

> *If multiple webpack runtimes (from different compilations) are used on the same webpage, there is a risk of conflicts of on-demand chunks in the global namespace.*

By default, on-demand chunk's output starts with:

example-on-demand-chunk.js

```
(window.webpackJsonp = window.webpackJsonp || []).push(/* ... */);
```

Change `output.jsonpFunction` for safe usage of multiple webpack runtimes on the same webpage:

webpack.config.flight-widget.js

```
module.exports = {
  //...
  output: {
    jsonpFunction: 'wpJsonpFlightsWidget'
  }
};
```

On-demand chunks content would now change to:

example-on-demand-chunk.js

```
(window.wpJsonpFlightsWidget = window.wpJsonpFlightsWidget || []).push(/* ... */);
```

output.library

`string` `object`

Can be given an `object` since webpack 3.1.0. Effective 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:

webpack.config.js

```
module.exports = {
  //...
  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.

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 be exposed using an `array` syntax (see this example for details).

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

output.libraryExport

`string [string]`

Configure which module or modules will be exposed via the `libraryTarget`. It is `undefined` by default, same behaviour will be applied if you set `libraryTarget` to an empty string e.g. `''` it will export the whole (namespace) object. The examples below demonstrate the effect of this config when using `libraryTarget: 'var'`.

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 = '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`.

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();
```

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.

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_));
```

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`.

> *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...

```
  module.exports = {
    //...
    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: 'amd-require'` - This packages your output with an immediately-executed AMD `require(dependencies, factory)` wrapper.

The `'amd-require'` target allows for the use of AMD dependencies without needing a separate later invocation. As with the `'amd'` target, this depends on the appropriate `require` function being available in the environment in which the webpack output is loaded.

With this target, the library name is ignored.

`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:

```
module.exports = {
  //...
  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:

```
module.exports = {
  //...
  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:

```
module.exports = {
  //...
  output: {
    library: {
      root: 'MyLibrary',
      amd: 'my-library',
      commonjs: 'my-common-library'
    },
    libraryTarget: 'umd'
  }
};
```

`libraryTarget: 'system'` - This will expose your library as a `System.register` module. This feature was first released in webpack 4.30.0.

System modules require that a global variable `System` is present in the browser when the webpack bundle is executed. Compiling to `System.register` format allows you to `System.import('/bundle.js')` without additional configuration and have your webpack bundle loaded into the System module registry.

```
module.exports = {
  //...
  output: {
```

```
    libraryTarget: 'system'
  }
```

Output:

```
System.register([], function(_export) {
  return {
    setters: [],
    execute: function() {
      // ...
    },
  };
});
```

By adding `output.library` to configuration in addition to having `output.libraryTarget` set to `system`, the output bundle will have the library name as an argument to `System.register`:

```
System.register('my-library', [], function(_export) {
  return {
    setters: [],
    execute: function() {
      // ...
    },
  };
});
```

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: path.join(process.cwd(), 'dist')`

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

webpack.config.js

```
module.exports = {
  //...
  output: {
    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`

Tells webpack to include comments in bundles with information about the contained modules. This option defaults to `true` in `development` and `false` in `production` mode respectively.

> *While the data this comments can provide is very useful during development when reading the generated code, it* ***should not*** *be used in production.*

webpack.config.js

```
module.exports = {
  //...
  output: {
    pathinfo: true
  }
};
```

> *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.

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

webpack.config.js

```
module.exports = {
  //...
  output: {
    path: path.resolve(__dirname, 'public/assets'),
    publicPath: 'https://cdn.example.com/assets/'
  }
};
```

For this configuration:

webpack.config.js

```
module.exports = {
  //...
  output: {
    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:

```
module.exports = {
  //...
  output: {
    // One of the below
    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 = '[file].map[query]'
```

Configure how source maps are named. Only takes effect when `devtool` is set to `'source-map'`, which writes an output file.

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	[file]
Description	The module filename

Template	[filebase]
Description	The module basename

output.sourcePrefix

`string = ''`

Change the prefix for each line in the output bundles.

webpack.config.js

```
module.exports = {
  //...
  output: {
    sourcePrefix: '\t'
  }
};
```

Using some kind of indentation makes bundles look prettier, but will cause issues with multi-line strings.

Typically you don't need to change `output.sourcePrefix`.

output.strictModuleExceptionHandling

`boolean = false`

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 throws
```

output.umdNamedDefine

`boolean`

When using `libraryTarget: "umd"`, setting `output.umdNamedDefine` to `true` will name the AMD module of the UMD build. Otherwise an anonymous `define` is used.

```
module.exports = {
  //...
  output: {
    umdNamedDefine: true
  }
};
```

output.futureEmitAssets

`boolean = false`

Tells webpack to use the future version of asset emitting logic, which allows freeing memory of assets after emitting. It could break plugins which assume that assets are still readable after they were emitted.

> *`output.futureEmitAssets` option will be removed in webpack v5.0.0 and this behaviour will become the new default.*

```
module.exports = {
  //...
  output: {
    futureEmitAssets: true
  }
};
```

Module

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

module.noParse

`RegExp` `[RegExp]` `function(resource)` `string` `[string]`

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.

webpack.config.js

```
module.exports = {
  //...
  module: {
    noParse: /jquery|lodash/,
  }
};
```

```
module.exports = {
  //...
  module: {
    noParse: (content) => /jquery|lodash/.test(content)
  }
};
```

module.rules

`[Rule]`

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.

module.unsafeCache

`boolean` `function (module)`

Cache the resolution of module requests, as well as the module's list of dependencies. Note that this feature can cause problems if the module is moved and should resolve to a different location. By default, this option is enabled only if `cache` is enabled.

webpack.config.js

```
module.exports = {
  //...
  module: {
    unsafeCache: false,
  }
};
```

In webpack 5, the default value is `true` if the `cache` option is enabled and the module appears to come from the node_modules directory.

Rule

`object`

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.

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`

```
string
```

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.

There are two phases that all loaders enter one after the other:

1. **Pitching** phase: the pitch method on loaders is called in the order `post, inline, normal, pre`. See Pitching Loader for details.
2. **Normal** phase: the normal method on loaders is executed in the order `pre, normal, inline, post`. Transformation on the source code of a module happens in this phase.

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.

```
// Disable normal loaders
import { a } from '!./file1.js';

// Disable preloaders and normal loaders
import { b } from  '-!./file2.js';

// Disable all loaders
import { c } from  '!!./file3.js';
```

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

*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.

webpack.config.js

```
module.exports = {
  //...
  module: {
    rules: [
      {
        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.

> *`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):

```
module.exports = {
  //...
  module: {
    rules: [
      {
        //...
        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
```

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:

webpack.config.js

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

Rule.rules

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

Rule.sideEffects

`bool`

Indicate what parts of the module contain side effects. See Tree Shaking for details.

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.type

`string`

Possible values: `'javascript/auto'` | `'javascript/dynamic'` | `'javascript/esm'` | `'json'` | `'webassembly/experimental'`

`Rule.type` sets the type for a matching module. This prevents defaultRules and their default importing behaviors from occurring. For example, if you want to load a `.json` file through a custom loader, you'd need to set the `type` to `javascript/auto` to bypass webpack's built-in json importing. (See v4.0 changelog for more details)

webpack.config.js

```
module.exports = {
  //...
  module: {
    rules: [
      //...
      {
        test: /\\.json$/,
        type: 'javascript/auto',
        loader: 'custom-json-loader'
      }
    ]
  }
};
```

Rule.use

`[UseEntry] function(info)`

`[UseEntry]`

`Rule.use` can be an array of UseEntry 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).

webpack.config.js

```
module.exports = {
  //...
  module: {
    rules: [
      {
        //...
        use: [
          'style-loader',
          {
            loader: 'css-loader',
            options: {
              importLoaders: 1
            }
          },
          {
            loader: 'less-loader',
            options: {
              noIeCompat: true
            }
          }
        ]
      }
    ]
  }
};
```

`function(info)`

`Rule.use` can also be a function which receives the object argument describing the module being loaded, and must return an array of `UseEntry` items.

The `info` object parameter has the following fields:

- `compiler` : The current webpack compiler (can be undefined)
- `issuer` : The path to the module that is importing the module being loaded
- `realResource` : Always the path to the module being loaded
- `resource` : The path to the module being loaded, it is usually equal to `realResource` except when the resource name is overwritten via `!=!` in request string

The same shortcut as an array can be used for the return value (i.e. `use: [ 'style-loader' ]`).

webpack.config.js

```
module.exports = {
  //...
  module: {
    rules: [
      {
        use: (info) => ([
          {
            loader: 'custom-svg-loader'
          },
          {
            loader: 'svgo-loader',
            options: {
              plugins: [{
                cleanupIDs: { prefix: basename(info.resource) }
              }]
            }
          }
        ])
      }
    ]
  }
};
```

See UseEntry for details.

`Rule.resolve`

> `Rule.resolve` *is Available since webpack 4.36.1*

Resolving can be configured on module level. See all available options on resolve configuration page. All applied resolve options get deeply merged with higher level resolve.

For example, let's imagine we have an entry in `./src/index.js`, `./src/footer/default.js` and a `./src/footer/overriden.js` to demonstrate the module level resolve.

./src/index.js

```
import footer from 'footer';
console.log(footer);
```

./src/footer/default.js

```
export default 'default footer';
```

./src/footer/overriden.js

```
export default 'overriden footer';
```

webpack.js.org

```
module.exports = {
  resolve: {
    alias: {
      'footer': './footer/default.js'
    }
  }
};
```

When creating a bundle with this configuration, `console.log(footer)` will output 'default footer'. Let's set `Rule.resolve` for `.js` files, and alias `footer` to `overriden.js`.

webpack.js.org

```
module.exports = {
  resolve: {
    alias: {
      'footer': './footer/default.js'
    }
  },
  module: {
    rules: [
      alias: {
        'footer': './footer/overriden.js'
      }
    ]
  }
};
```

When creating a bundle with updated configuration, `console.log(footer)` will output 'overriden footer'.

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:

```
module.exports = {
  //...
  module: {
    rules: [
      {
        test: /\\.css$/,
        include: [
          path.resolve(__dirname, 'app/styles'),
          path.resolve(__dirname, 'vendor/styles')
        ]
      }
    ]
```

UseEntry

`object` `function(info)`

`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.

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.

webpack.config.js

```
module.exports = {
  //...
  module: {
    rules: [
      {
        loader: 'css-loader',
        options: {
          modules: true
        }
      }
    ]
  }
};
```

`function(info)`

A `UseEntry` can also be a function which receives the object argument describing the module being loaded, and must return an options object. This can be used to vary the loader options on a per-module basis.

The `info` object parameter has the following fields:

- `compiler` : The current webpack compiler (can be undefined)
- `issuer` : The path to the module that is importing the module being loaded
- `realResource` : Always the path to the module being loaded
- `resource` : The path to the module being loaded, it is usually equal to `realResource` except when the resource name is overwritten via `!=!` in request string

webpack.config.js

```
module.exports = {
  //...
  module: {
    rules: [
      {
        loader: 'file-loader',
        options: {
          outputPath: 'svgs'
        }
      },
      (info) => ({
        loader: 'svgo-loader',
        options: {
          plugins: [{
            cleanupIDs: { prefix: basename(info.resource) }
          }]
        }
      })
    ]
  }
};
```

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:

webpack.config.js

```
module.exports = {
  //...
  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
  }
};
```

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

Resolve

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`).

webpack.config.js

```
module.exports = {
  //...
  resolve: {
    // configuration options
  }
};
```

resolve.alias

`object`

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

webpack.config.js

```
module.exports = {
  //...
  resolve: {
    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:

webpack.config.js

```
module.exports = {
  //...
  resolve: {
    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
alias: import 'xyz' import 'xyz/file.js' error	{ xyz: '/abs/path/to/file.js' } /abs/path/to/file.js
alias: import 'xyz' import 'xyz/file.js'	{ xyz$: '/abs/path/to/file.js' } /abs/path/to/file.js /abc/node_modules/xyz/file.js
alias: import 'xyz'	{ xyz: './dir/file.js' } /abc/dir/file.js

import 'xyz/file.js'	
error	

alias:	{ xyz$: './dir/file.js' }
import 'xyz'	/abc/dir/file.js
import 'xyz/file.js'	
/abc/node_modules/xyz/file.js	

alias:	{ xyz: '/some/dir' }
import 'xyz'	/some/dir/index.js
import 'xyz/file.js'	
/some/dir/file.js	

alias:	{ xyz$: '/some/dir' }
import 'xyz'	/some/dir/index.js
import 'xyz/file.js'	
/abc/node_modules/xyz/file.js	

alias:	{ xyz: './dir' }
import 'xyz'	/abc/dir/index.js
import 'xyz/file.js'	
/abc/dir/file.js	

alias:	{ xyz: 'modu' }
import 'xyz'	/abc/node_modules/modu/index.js
import 'xyz/file.js'	
/abc/node_modules/modu/file.js	

alias:	{ xyz$: 'modu' }
import 'xyz'	/abc/node_modules/modu/index.js
import 'xyz/file.js'	
/abc/node_modules/xyz/file.js	

alias:	{ xyz: 'modu/some/file.js' }
import 'xyz'	/abc/node_modules/modu/some/file.js
import 'xyz/file.js'	

error
alias: { xyz: 'modu/dir' } import 'xyz' /abc/node_modules/modu/dir/index.js import 'xyz/file.js' /abc/node_modules/dir/file.js
alias: { xyz: 'xyz/dir' } import 'xyz' /abc/node_modules/xyz/dir/index.js import 'xyz/file.js' /abc/node_modules/xyz/dir/file.js
alias: { xyz$: 'xyz/dir' } import 'xyz' /abc/node_modules/xyz/dir/index.js import 'xyz/file.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.alias` takes precedence over other module resolutions.

resolve.aliasFields

`[string]: ['browser']`

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

webpack.config.js

```
module.exports = {
  //...
  resolve: {
    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

`[string] = ['package.json']`

The JSON files to use for descriptions.

webpack.config.js

```
module.exports = {
  //...
  resolve: {
    descriptionFiles: ['package.json']
  }
};
```

resolve.enforceExtension

`boolean = false`

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.

webpack.config.js

```
module.exports = {
  //...
  resolve: {
    enforceExtension: false
  }
};
```

resolve.enforceModuleExtension

`boolean = false`

Removed in webpack 5

Tells webpack whether to require to use an extension for modules (e.g. loaders).

webpack.config.js

```
module.exports = {
  //...
  resolve: {
    enforceModuleExtension: false
  }
};
```

resolve.extensions

`[string] = ['.wasm', '.mjs', '.js', '.json']`

Attempt to resolve these extensions in order.

> *If multiple files share the same name but have different extensions, webpack will resolve the one with the extension listed first in the array and skip the rest.*

webpack.config.js

```
module.exports = {
  //...
  resolve: {
    extensions: ['.wasm', '.mjs', '.js', '.json']
  }
};
```

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

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

> *Using this will **override the default array**, meaning that webpack will no longer try to resolve modules using the default extensions.*

resolve.mainFields

`[string]`

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:

webpack.config.js

```
module.exports = {
  //...
  resolve: {
    mainFields: ['browser', 'module', 'main']
  }
};
```

For any other target (including `node`):

webpack.config.js

```
module.exports = {
  //...
  resolve: {
    mainFields: ['module', 'main']
  }
};
```

For example, consider an arbitrary library called `upstream` with a `package.json` that contains the following fields:

```
{
  "browser": "build/upstream.js",
  "module": "index"
}
```

When we `import * as Upstream from 'upstream'` this will actually resolve to the file in the `browser` property. The `browser` property takes precedence because it's the first item in `mainFields`. Meanwhile, a Node.js application bundled by webpack will first try to resolve using the file in the `module` field.

`resolve.mainFiles`

`[string] = ['index']`

The filename to be used while resolving directories.

webpack.config.js

```
module.exports = {
  //...
  resolve: {
    mainFiles: ['index']
  }
};
```

resolve.modules

`[string] = ['node_modules']`

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.

webpack.config.js

```
module.exports = {
  //...
  resolve: {
    modules: ['node_modules']
  }
};
```

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

webpack.config.js

```
module.exports = {
  //...
  resolve: {
    modules: [path.resolve(__dirname, 'src'), 'node_modules']
  }
};
```

resolve.unsafeCache

`RegExp [RegExp] boolean: true`

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

webpack.config.js

```
module.exports = {
  //...
  resolve: {
    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:

webpack.config.js

```
module.exports = {
  //...
  resolve: {
    unsafeCache: /src\/utilities/
  }
};
```

Changes to cached paths may cause failure in rare cases.

resolve.plugins

`[Plugin]`

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

webpack.config.js

```
module.exports = {
  //...
  resolve: {
    plugins: [
      new DirectoryNamedWebpackPlugin()
    ]
  }
};
```

resolve.symlinks

`boolean = true`

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`).

webpack.config.js

```
module.exports = {
  //...
  resolve: {
    symlinks: true
  }
};
```

resolve.cachePredicate

`function(module) => boolean`

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

webpack.config.js

```
module.exports = {
  //...
  resolve: {
    cachePredicate: (module) => {
      // additional logic
      return true;
    }
  }
};
```

resolveLoader

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

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

webpack.config.js

```
module.exports = {
  //...
  resolveLoader: {
    modules: ['node_modules'],
    extensions: ['.js', '.json'],
    mainFields: ['loader', 'main']
  }
};
```

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`.

resolve.roots

`[string]`

A list of directories where requests of server-relative URLs (starting with '/') are resolved, defaults to `context` configuration option. On non-Windows systems these requests are resolved as an absolute path first.

webpack.config.js

```
const fixtures = path.resolve(__dirname, 'fixtures');
module.exports = {
  //...
  resolve: {
    roots: [__dirname, fixtures]
  }
};
```

resolveLoader.moduleExtensions

`[string]`

Removed in webpack 5

The extensions/suffixes 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:

webpack.config.js

```
module.exports = {
  //...
  resolveLoader: {
    moduleExtensions: ['-loader']
  }
};
```

Optimization

Since version 4 webpack runs optimizations for you depending on the chosen `mode`, still all optimizations are available for manual configuration and overrides.

`optimization.minimize`

`boolean`

Tell webpack to minimize the bundle using the TerserPlugin.

This is `true` by default in `production` mode.

webpack.config.js

```
module.exports = {
  //...
  optimization: {
    minimize: false
  }
};
```

Learn how mode works.

`optimization.minimizer`

`[Plugin]` and or `[function (compiler)]`

Allows you to override the default minimizer by providing a different one or more customized TerserPlugin instances.

webpack.config.js

```
const TerserPlugin = require('terser-webpack-plugin');

module.exports = {
  optimization: {
    minimizer: [
      new TerserPlugin({
        cache: true,
        parallel: true,
        sourceMap: true, // Must be set to true if using source-maps in production
        terserOptions: {
          // https://github.com/webpack-contrib/terser-webpack-plugin#terseroptions
        }
      }),
    ],
  }
};
```

Or, as function:

```
module.exports = {
  optimization: {
    minimizer: [
      (compiler) => {
        const TerserPlugin = require('terser-webpack-plugin');
        new TerserPlugin({ /* your config */ }).apply(compiler);
      }
    ],
  }
};
```

optimization.splitChunks

`object`

By default webpack v4+ provides new common chunks strategies out of the box for dynamically imported modules. See available options for configuring this behavior in the SplitChunksPlugin page.

optimization.runtimeChunk

`object` `string` `boolean`

Setting `optimization.runtimeChunk` to `true` or `'multiple'` adds an additional chunk to each entrypoint containing only the runtime. This setting is an alias for:

webpack.config.js

```
module.exports = {
  //...
  optimization: {
    runtimeChunk: {
      name: entrypoint => `runtime~${entrypoint.name}`
    }
  }
};
```

The value `'single'` instead creates a runtime file to be shared for all generated chunks. This setting is an alias for:

webpack.config.js

```
module.exports = {
  //...
  optimization: {
    runtimeChunk: {
      name: 'runtime'
    }
  }
};
```

By setting `optimization.runtimeChunk` to `object` it is only possible to provide the `name` property which stands for the name or name factory for the runtime chunks.

Default is `false` : each entry chunk embeds runtime.

> *Imported modules are initialized for each runtime chunk separately, so if you include multiple entry points on a page, beware of this behavior. You will probably want to set it to `single` or use another configuration that allows you to only have one runtime instance.*

webpack.config.js

```
module.exports = {
  //...
```

```
  optimization: {
    runtimeChunk: {
      name: entrypoint => `runtimechunk~${entrypoint.name}`
    }
  }
};
```

optimization.noEmitOnErrors

`boolean`

Use the `optimization.noEmitOnErrors` to skip the emitting phase whenever there are errors while compiling. This ensures that no erroring assets are emitted. The `emitted` flag in the stats is `false` for all assets.

webpack.config.js

```
module.exports = {
  //...
  optimization: {
    noEmitOnErrors: true
  }
};
```

If you are using webpack CLI, the webpack process will not exit with an error code while this plugin is enabled. If you want webpack to "fail" when using the CLI, please check out the `bail` option.

optimization.namedModules

`boolean = false`

Tells webpack to use readable module identifiers for better debugging. When `optimization.namedModules` is not set in webpack config, webpack will enable it by default for mode `development` and disable for mode `production` .

webpack.config.js

```
module.exports = {
  //...
  optimization: {
    namedModules: true
```

optimization.namedChunks

`boolean = false`

Tells webpack to use readable chunk identifiers for better debugging. This option is enabled by default for mode `development` and disabled for mode `production` if no option is provided in webpack config.

webpack.config.js

```
module.exports = {
  //...
  optimization: {
    namedChunks: true
  }
};
```

optimization.moduleIds

`boolean = false  string: 'natural' | 'named' | 'hashed' | 'size' | 'total-size'`

Tells webpack which algorithm to use when choosing module ids. Setting `optimization.moduleIds` to `false` tells webpack that none of built-in algorithms should be used, as custom one can be provided via plugin. By default `optimization.moduleIds` is set to `false` .

The following string values are supported:

Option	`natural`
Description	Numeric ids in order of usage.
Option	`named`
Description	Readable ids for better debugging.
Option	`hashed`
Description	Short hashes as ids for better long term caching.
Option	`size`

Description	Numeric ids focused on minimal initial download size.

Option Description	`total-size` numeric ids focused on minimal total download size.

webpack.config.js

```
module.exports = {
  //...
  optimization: {
    moduleIds: 'hashed'
  }
};
```

optimization.chunkIds

`boolean = false` `string: 'natural' | 'named' | 'size' | 'total-size'`

Tells webpack which algorithm to use when choosing chunk ids. Setting `optimization.chunkIds` to `false` tells webpack that none of built-in algorithms should be used, as custom one can be provided via plugin. There are couple of defaults for `optimization.chunkIds`:

- if `optimization.occurrenceOrder` is enabled `optimization.chunkIds` is set to `'total-size'`
- Disregarding previous if, if `optimization.namedChunks` is enabled `optimization.chunkIds` is set to `'named'`
- if none of the above, `optimization.chunkIds` will be defaulted to `'natural'`

The following string values are supported:

Option Description	`'natural'` Numeric ids in order of usage.

Option Description	`'named'` Readable ids for better debugging.

Option	`'size'`

Description	Numeric ids focused on minimal initial download size.
Option Description	`'total-size'` numeric ids focused on minimal total download size.

webpack.config.js

```
module.exports = {
  //...
  optimization: {
    chunkIds: 'named'
  }
};
```

optimization.nodeEnv

`boolean = false  string`

Tells webpack to set `process.env.NODE_ENV` to a given string value. `optimization.nodeEnv` uses DefinePlugin unless set to `false`. `optimization.nodeEnv` **defaults** to mode if set, else falls back to `'production'`.

Possible values:

- any string: the value to set `process.env.NODE_ENV` to.
- false: do not modify/set the value of `process.env.NODE_ENV`.

webpack.config.js

```
module.exports = {
  //...
  optimization: {
    nodeEnv: 'production'
  }
};
```

optimization.mangleWasmImports

`boolean = false`

When set to `true` tells webpack to reduce the size of WASM by changing imports to shorter strings. It mangles module and export names.

webpack.config.js

```
module.exports = {
  //...
  optimization: {
    mangleWasmImports: true
  }
};
```

optimization.removeAvailableModules

`boolean = false`

Tells webpack to detect and remove modules from chunks when these modules are already included in all parents. Setting `optimization.removeAvailableModules` to `true` will enable this optimization. Enabled by default in `production` mode.

webpack.config.js

```
module.exports = {
  //...
  optimization: {
    removeAvailableModules: true
  }
};
```

> *`optimization.removeAvailableModules` reduces the performance of webpack, and will be disabled in `production` mode by default in next major release. Disable it in `production` mode if you want extra build performance.*

optimization.removeEmptyChunks

`boolean = true`

Tells webpack to detect and remove chunks which are empty. Setting `optimization.removeEmptyChunks` to `false` will disable this optimization.

webpack.config.js

```
module.exports = {
  //...
  optimization: {
    removeEmptyChunks: false
  }
};
```

optimization.mergeDuplicateChunks

`boolean = true`

Tells webpack to merge chunks which contain the same modules. Setting `optimization.mergeDuplicateChunks` to `false` will disable this optimization.

webpack.config.js

```
module.exports = {
  //...
  optimization: {
    mergeDuplicateChunks: false
  }
};
```

optimization.flagIncludedChunks

`boolean`

Tells webpack to determine and flag chunks which are subsets of other chunks in a way that subsets don't have to be loaded when the bigger chunk has been already loaded. By default `optimization.flagIncludedChunks` is enabled in `production` mode and disabled elsewise.

webpack.config.js

```
module.exports = {
  //...
  optimization: {
    flagIncludedChunks: true
  }
};
```

optimization.occurrenceOrder

`boolean`

Tells webpack to figure out an order of modules which will result in the smallest initial bundle. By default `optimization.occurrenceOrder` is enabled in `production` mode and disabled elsewise.

webpack.config.js

```
module.exports = {
  //...
  optimization: {
    occurrenceOrder: false
  }
};
```

optimization.providedExports

`boolean`

Tells webpack to figure out which exports are provided by modules to generate more efficient code for `export * from ...`. By default `optimization.providedExports` is enabled.

webpack.config.js

```
module.exports = {
  //...
  optimization: {
    providedExports: false
  }
};
```

optimization.usedExports

`boolean`

Tells webpack to determine used exports for each module. This depends on `optimization.providedExports`. Information collected by `optimization.usedExports` is used by other optimizations or code generation i.e. exports are not generated for unused exports, export names are mangled to single char identifiers when all usages are compatible. Dead code elimination in minimizers will benefit from this and can remove unused exports. By default `optimization.usedExports` is enabled in `production` mode and disabled elsewise.

webpack.config.js

```
module.exports = {
  //...
  optimization: {
    usedExports: true
  }
};
```

optimization.concatenateModules

`boolean`

Tells webpack to find segments of the module graph which can be safely concatenated into a single module. Depends on `optimization.providedExports` and `optimization.usedExports`. By default `optimization.concatenateModules` is enabled in `production` mode and disabled elsewise.

webpack.config.js

```
module.exports = {
  //...
  optimization: {
    concatenateModules: true
  }
};
```

optimization.sideEffects

`boolean`

Tells webpack to recognise the `sideEffects` flag in `package.json` or rules to skip over modules which are flagged to contain no side effects when exports are not used.

package.json

```
{
  "name": "awesome npm module",
  "version": "1.0.0",
  "sideEffects": false
}
```

Please note that `sideEffects` should be in the npm module's `package.json` file and doesn't mean that you need to set `sideEffects` to `false` in your own project's `package.json` which requires that big module.

`optimization.sideEffects` depends on `optimization.providedExports` to be enabled. This dependency has a build time cost, but eliminating modules has positive impact on performance because of less code generation. Effect of this optimization depends on your codebase, try it for possible performance wins.

By default `optimization.sideEffects` is enabled in `production` mode and disabled elsewise.

webpack.config.js

```
module.exports = {
  //...
  optimization: {
    sideEffects: true
  }
};
```

optimization.portableRecords

`boolean`

`optimization.portableRecords` tells webpack to generate records with relative paths to be able to move the context folder.

By default `optimization.portableRecords` is disabled. Automatically enabled if at least one of the records options provided to webpack config: `recordsPath`, `recordsInputPath`, `recordsOutputPath`.

webpack.config.js

```
module.exports = {
  //...
  optimization: {
    portableRecords: true
  }
};
```

Plugins

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 Plugins page for a list of plugins and documentation but note that there are a lot more out in the community.

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

`plugins`

`[Plugin]`

An array of webpack plugins. For example, `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.

webpack.config.js

```
module.exports = {
  //...
  plugins: [
    new webpack.DefinePlugin({
      // Definitions...
    })
  ]
};
```

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

webpack.config.js

```
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
module.exports = {
  //...
  plugins: [
    new ExtractTextPlugin({
      filename: 'build.min.css',
```

```
    allChunks: true,
  }),
      b   k I     Pl  i (/^\ \/l   l $/  /     t$/)
```

DevServer

webpack-dev-server can be used to quickly develop an application. See the development guide to get started.

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

> *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 in the project root:

webpack.config.js

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

module.exports = {
  //...
  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.

> *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.*

> *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`.*

> *HTML template is required to serve the bundle, usually it is an `index.html` file. Make sure that script references are added into HTML, webpack-dev-server doesn't inject them automatically.*

devServer.after

`function (app, server, compiler)`

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

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    after: function(app, server, compiler) {
      // do fancy stuff
    }
  }
};
```

devServer.allowedHosts

`[string]`

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

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    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`.

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    // 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.c
```

devServer.before

`function (app, server, compiler)`

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:

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    before: function(app, server, compiler) {
      app.get('/some/path', function(req, res) {
        res.json({ custom: 'response' });
      });
    }
  }
};
```

devServer.bonjour

`boolean = false`

This option broadcasts the server via ZeroConf networking on start

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    bonjour: true
  }
};
```

Usage via the CLI

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

devServer.clientLogLevel

`string = 'info': 'silent' | 'trace' | 'debug' | 'info' | 'warn' | 'error' | 'none' | 'warning'`

`none` and `warning` are going to be deprecated at the next major version.

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.

`devServer.clientLogLevel` may be too verbose, you can turn logging off by setting it to `'silent'`.

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    clientLogLevel: 'silent'
  }
};
```

Usage via the CLI

```
webpack-dev-server --client-log-level silent
```

`devServer.color` - CLI only

`boolean`

Enables/Disables colors on the console.

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

`devServer.compress`

`boolean`

Enable gzip compression for everything served:

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    compress: true
  }
};
```

Usage via the CLI

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

devServer.contentBase

`boolean: false  string  [string]  number`

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.

It is recommended to use an absolute path.

By default it will use your current working directory to serve content. To disable `contentBase` set it to `false`.

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    contentBase: path.join(__dirname, 'public')
  }
};
```

It is also possible to serve from multiple directories:

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    contentBase: [path.join(__dirname, 'public'), path.join(__dirname, 'assets')]
  }
};
```

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.

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    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 `devServer.filename` is used like this:

webpack.config.js

```
module.exports = {
  //...
  output: {
    filename: 'bundle.js'
  },
  devServer: {
    lazy: true,
    filename: 'bundle.js'
  }
};
```

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

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

devServer.headers 🔑

`object`

Adds headers to all responses:

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    headers: {
      'X-Custom-Foo': 'bar'
    }
  }
};
```

devServer.historyApiFallback

`boolean = false` `object`

When using the HTML5 History API, the `index.html` page will likely have to be served in place of any `404` responses. Enable `devServer.historyApiFallback` by setting it to `true`:

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    historyApiFallback: true
  }
};
```

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

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    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`:

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    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 = 'localhost'
```

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

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    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:

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    hot: true
  }
};
```

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.

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    hotOnly: true
  }
};
```

Usage via the CLI

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

devServer.http2

```
boolean = false
```

Serve over HTTP/2 using spdy. This option is ignored for Node 10.0.0 and above, as spdy is broken for those versions. The dev server will migrate over to Node's built-in HTTP/2 once Express supports it.

If `devServer.http2` is not explicitly set to `false`, it will default to `true` when `devServer.https` is enabled. When `devServer.http2` is enabled but the server is unable to serve over HTTP/2, the server defaults to HTTPS.

HTTP/2 with a self-signed certificate:

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    http2: true
  }
};
```

Provide your own certificate using the https option:

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    http2: true,
    https: {
      key: fs.readFileSync('/path/to/server.key'),
      cert: fs.readFileSync('/path/to/server.crt'),
      ca: fs.readFileSync('/path/to/ca.pem'),
    }
  }
};
```

Usage via CLI

```
webpack-dev-server --http2
```

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

```
webpack-dev-server --http2 --key /path/to/server.key --cert /path/to/server.crt --cacert /path/to/c
```

devServer.https

`boolean object`

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

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    https: true
  }
};
```

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

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    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/c
```

devServer.index

`string`

The filename that is considered the index file.

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    index: 'index.html'
  }
};
```

`devServer.info` - CLI only

`boolean`

Output cli information. It is enabled by default.

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

`devServer.injectClient`

`boolean = false  function (compilerConfig) => boolean`

Tells `devServer` to inject a client. Setting `devServer.injectClient` to `true` will result in always injecting a client. It is possible to provide a function to inject conditionally:

```
module.exports = {
  //...
  devServer: {
    injectClient: (compilerConfig) => compilerConfig.name === 'only-include'
  }
};
```

`devServer.injectHot`

`boolean = false  function (compilerConfig) => boolean`

Tells `devServer` to inject a Hot Module Replacement. Setting `devServer.injectHot` to `true` will result in always injecting. It is possible to provide a function to inject conditionally:

```
module.exports = {
  //...
  devServer: {
    hot: true,
    injectHot: (compilerConfig) => compilerConfig.name === 'only-include'
  }
};
```

Make sure that `devServer.hot` is set to `true` because `devServer.injectHot` only works with HMR.

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**:

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    inline: false
  }
};
```

Usage via the CLI

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

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 `devServer.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**.

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    lazy: true
  }
};
```

Usage via the CLI

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

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

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

devServer.liveReload

`boolean = true`

By default, the dev-server will reload/refresh the page when file changes are detected. `devServer.hot` option must be disabled or `devServer.watchContentBase` option must be enabled in order for `liveReload` to take effect. Disable `devServer.liveReload` by setting it to `false`:

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    liveReload: false
  }
};
```

Usage via the CLI

```
webpack-dev-server --no-live-reload
```

devServer.mimeTypes 

`object`

Allows dev-server to register custom mime types. The object is passed to the underlying `webpack-dev-middleware`. See documentation for usage notes.

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    mimeTypes: { 'text/html': ['phtml'] }
  }
};
```

devServer.noInfo 

`boolean = false`

Tells dev-server to supress messages like the webpack bundle information. Errors and warnings will still be shown.

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    noInfo: true
  }
};
```

devServer.onListening

`function (server)`

Provides an option to execute a custom function when `webpack-dev-server` starts listening for connections on a port.

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    onListening: function(server) {
      const port = server.listeningApp.address().port;
      console.log('Listening on port:', port);
    }
  }
};
```

devServer.open

`boolean: false  string`

Tells dev-server to open the browser after server had been started. Set it to `true` to open your default browser.

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    open: true
  }
};
```

Provide browser name to use instead of the default one:

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    open: 'Google Chrome'
  }
};
```

Usage via the CLI

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

The browser application name is platform dependent. Don't hard code it in reusable modules. For example, `'Chrome'` is `'Google Chrome'` on macOS, `'google-chrome'` on Linux and `'chrome'` on Windows.

devServer.openPage

`string  [string]`

Specify a page to navigate to when opening the browser.

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    openPage: '/different/page'
  }
};
```

Usage via the CLI

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

If you wish to specify multiple pages to open in the browser.

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    openPage: ['/different/page1', '/different/page2']
  }
};
```

Usage via the CLI

```
webpack-dev-server --open-page "/different/page1,/different/page2"
```

devServer.overlay

`boolean = false  object: { errors boolean = false, warnings boolean = false }`

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

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    overlay: true
  }
};
```

If you want to show warnings as well as errors:

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    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.

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    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.

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    pfxPassphrase: 'passphrase'
  }
};
```

Usage via the CLI

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

devServer.port

`number`

Specify a port number to listen for requests on:

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    port: 8080
  }
};
```

Usage via the CLI

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

devServer.proxy

`object  [object, function]`

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. Check out its documentation for more advanced usages. Note that some of `http-proxy-middleware` 's features do not require a `target` key, e.g. its `router` feature, but you will still need to include a `target` key in your config here, otherwise `webpack-dev-server` won't pass it along to `http-proxy-middleware`).

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

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    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:

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    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:

webpack.config.js

```
module.exports = {
  //...
  devServer: {
```

```
    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.

- Return `null` or `undefined` to continue processing the request with proxy.
- Return `false` to produce a 404 error for the request.
- Return a path to serve from, 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:

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    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:

webpack.config.js

```
module.exports = {
  //...
```

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

Note that requests to root won't be proxied by default. To enable root proxying, the `devServer.index` option should be specified as a falsy value:

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    index: '', // specify to enable root proxying
    host: '...',
    contentBase: '...',
    proxy: {
      context: () => true,
      target: 'http://localhost:1234'
    }
  }
};
```

The origin of the host header is kept when proxying by default, you can set `changeOrigin` to `true` to override this behaviour. It is useful in some cases like using name-based virtual hosted sites.

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    proxy: {
      '/api': {
        target: 'http://localhost:3000',
        changeOrigin: true
      }
    }
  }
};
```

`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`:

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    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 `devServer.publicPath` is `'/'`, so your bundle is available as `http://localhost:8080/bundle.js`.

Change `devServer.publicPath` to put bundle under specific directory:

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    publicPath: '/assets/'
  }
};
```

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

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

It is also possible to use a full URL.

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    publicPath: 'http://localhost:8080/assets/'
  }
};
```

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

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

`devServer.quiet` 🔑

`boolean`

With `devServer.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.

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    quiet: true
  }
};
```

Usage via the CLI

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

devServer.serveIndex

`boolean = true`

Tells dev-server to use `serveIndex` middleware when enabled.

`serveIndex` middleware generates directory listings on viewing directories that don't have an index.html file.

```
module.exports = {
  //...
  devServer: {
    serveIndex: true
  }
};
```

devServer.setup

`function (app, server)`

*This option is **deprecated** in favor of `devServer.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:

webpack.config.js

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

devServer.socket

`string`

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

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    socket: 'socket'
  }
};
```

Usage via the CLI

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

devServer.sockHost

`string`

Tells clients connected to `devServer` to use provided socket host.

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    sockHost: 'myhost.test'
  }
};
```

devServer.sockPath

`string = '/sockjs-node'`

The path at which to connect to the reloading socket.

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    sockPath: '/socket',
  }
};
```

Usage via the CLI

```
webpack-dev-server --sockPath /socket
```

devServer.sockPort

`number` `string`

Tells clients connected to `devServer` to use provided socket port.

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    sockPort: 8080
  }
};
```

devServer.staticOptions

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

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    staticOptions: {
      redirect: false
    }
  }
};
```

This only works when using `devServer.contentBase` as a `string`.

`devServer.stats` 🔑

`string: 'none' | 'errors-only' | 'minimal' | 'normal' | 'verbose'  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:

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    stats: 'errors-only'
  }
};
```

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

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.transportMode`

`string = 'sockjs': 'sockjs' | 'ws'  object`

`transportMode` is an experimental option, meaning its usage could potentially change without warning.

Providing a string to `devServer.transportMode` is a shortcut to setting both `devServer.transportMode.client` and `devServer.transportMode.server` to the given string value.

This option allows us either to choose the current `devServer` transport mode for client/server individually or to provide custom client/server implementation. This allows to specify how browser or other client communicates with the `devServer`.

The current default mode is `'sockjs'`. This mode uses SockJS-node as a server, and SockJS-client on the client.

`'ws'` mode will become the default mode in the next major `devServer` version. This mode uses ws as a server, and native WebSockets on the client.

Use `'ws'` mode:

```
module.exports = {
  //...
  devServer: {
    transportMode: 'ws'
  }
};
```

When providing a custom client and server implementation make sure that they are compatible with one another to communicate successfully.

devServer.transportMode.client

`string` `path`

To create a custom client implementation, create a class that extends `BaseClient`.

Using path to `CustomClient.js`, a custom WebSocket client implementation, along with the compatible `'ws'` server:

```
module.exports = {
  //...
  devServer: {
    transportMode: {
      client: require.resolve('./CustomClient'),
      server: 'ws'
    }
```

devServer.transportMode.server

`string` `path` `function`

To create a custom server implementation, create a class that extends `BaseServer`.

Using path to `CustomServer.js`, a custom WebSocket server implementation, along with the compatible `'ws'` client:

```
module.exports = {
  //...
  devServer: {
    transportMode: {
      client: 'ws',
      server: require.resolve('./CustomServer')
    }
  }
};
```

Using class exported by `CustomServer.js`, a custom WebSocket server implementation, along with the compatible `'ws'` client:

```
module.exports = {
  //...
  devServer: {
    transportMode: {
      client: 'ws',
      server: require('./CustomServer')
    }
  }
};
```

Using custom, compatible WebSocket client and server implementations:

```
module.exports = {
  //...
  devServer: {
    transportMode: {
      client: require.resolve('./CustomClient'),
      server: require.resolve('./CustomServer')
    }
  }
};
```

devServer.useLocalIp

`boolean`

This option lets the browser open with your local IP.

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    useLocalIp: true
  }
};
```

Usage via the CLI

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

devServer.watchContentBase

`boolean`

Tell dev-server to watch the files served by the `devServer.contentBase` option. It is disabled by default. When enabled, file changes will trigger a full page reload.

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    watchContentBase: true
  }
};
```

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:

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    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.

devServer.writeToDisk 🔑

`boolean = false  function (filePath)`

Tells `devServer` to write generated assets to the disk. The output is written to the output.path directory.

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    writeToDisk: true
  }
};
```

Providing a `Function` to `devServer.writeToDisk` can be used for filtering. The function follows the same premise as `Array#filter` in which a boolean return value tells if the file should be written to disk.

webpack.config.js

```
module.exports = {
  //...
  devServer: {
    writeToDisk: (filePath) => {
```

```
    return /superman\.css$/.test(filePath);
}
```

Devtool

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.

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

> *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	(none)
build	fastest
rebuild	
production	
quality	
fastest	
yes	
bundled code	

devtool build rebuild production quality	eval fastest
fastest	
no	
generated code	

devtool build rebuild production quality	cheap-eval-source-map fast
faster	
no	
transformed code (lines only)	

devtool build rebuild production quality	cheap-module-eval- source-map slow
faster	
no	
original source (lines only)	

devtool build rebuild production quality	eval-source-map slowest
fast	
no	
original source	

devtool build rebuild production quality	cheap-source-map fast slow yes
transformed code (lines only)	
devtool build rebuild production quality	cheap-module-source-map slow slower yes
original source (lines only)	
devtool build rebuild production quality	inline-cheap-source-map fast slow no
transformed code (lines only)	
devtool build rebuild production quality	inline-cheap-module-source-map slow slower no
original source (lines only)	

devtool	source-map
build	slowest
rebuild	
production	
quality	
slowest	
yes	
original source	

devtool	inline-source-map
build	slowest
rebuild	
production	
quality	
slowest	
no	
original source	

devtool	hidden-source-map
build	slowest
rebuild	
production	
quality	
slowest	
yes	
original source	

devtool	nosources-source-map
build	slowest
rebuild	
production	
quality	
slowest	
yes	
without source content	

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.

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

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 it 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.

> *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.

> *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.

> *It still exposes filenames and structure for decompiling, but it doesn't expose the original code.*

> *When using the `terser-webpack-plugin` you must provide the `sourceMap: true` option to enable SourceMap support.*

Target

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	`async-node`
Description	Compile for usage in a Node.js-like environment (uses `fs` and `vm` to load chunks asynchronously)

Option Description	`electron-main` Compile for Electron for main process.
Option Description	`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.
Option Description	`electron-preload` Compile for Electron for renderer process, providing a target using `NodeTemplatePlugin` with `asyncChunkLoading` set to `true`, `FunctionModulePlugin` for browser environments and `NodeTargetPlugin` and `ExternalsPlugin` for CommonJS and Electron built-in modules.
Option Description	`node` Compile for usage in a Node.js-like environment (uses Node.js `require` to load chunks)
Option Description	`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)
Option Description	`web` Compile for usage in a browser-like environment **(default)**
Option Description	`webworker` Compile as WebWorker

For example, when the `target` is set to `"electron-main"`, 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 `target` 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 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')
    );
  }
};
```

Watch and WatchOptions

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 = false`

Turn on watch mode. This means that after the initial build, webpack will continue to watch for changes in any of the resolved files.

webpack.config.js

```
module.exports = {
  //...
  watch: true
};
```

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:

webpack.config.js

```
module.exports = {
  //...
  watchOptions: {
    aggregateTimeout: 300,
    poll: 1000
  }
};
```

watchOptions.aggregateTimeout

`number = 300`

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:

```
module.exports = {
  //...
  watchOptions: {
    aggregateTimeout: 600
  }
};
```

watchOptions.ignored

`RegExp` `anymatch`

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`:

webpack.config.js

```
module.exports = {
  //...
  watchOptions: {
    ignored: /node_modules/
  }
};
```

It is also possible to have and use multiple anymatch patterns:

webpack.config.js

```
module.exports = {
  //...
  watchOptions: {
    ignored: ['files/**/*.js', 'node_modules']
  }
};
```

If you use `require.context`, webpack will watch your entire directory. You will need to ignore files and/or directories so that unwanted changes will not trigger a rebuild.

watchOptions.poll

`boolean = false  number`

Turn on polling by passing `true`, or specifying a poll interval in milliseconds:

webpack.config.js

```
module.exports = {
  //...
  watchOptions: {
    poll: 1000 // Check for changes every second
  }
};
```

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|Preferences} > Appearance & Behavior > System Settings > Use "safe write" (save changes to a temporary file first)`.

Externals

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.

> *consumer here is any end-user application.*

`externals`

```
string  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

```
module.exports = {
  //...
  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

```
module.exports = {
  //...
  externals: {
    subtract: ['./math', 'subtract']
  }
};
```

`subtract: ['./math', 'subtract']` allows you select part of a commonjs module, where `./math` is the module and your bundle only requires the subset under the `subtract` variable. This example would translate to `require('./math').subtract;`

object

An object with `{ root, amd, commonjs, ... }` is only allowed for `libraryTarget: 'umd'`. It's not allowed for other library targets.

```
module.exports = {
  //...
  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 too, for example, whitelist packages.

It basically comes down to this:

```
module.exports = {
  //...
  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.

```
module.exports = {
  //...
  externals: /^(jquery|\$)$/i
};
```

In this case, any dependency named `jQuery` , capitalized or not, or `$` would be externalized.

Combining syntaxes

Sometimes you may want to use a combination of the above syntaxes. This can be done in the following manner:

```
module.exports = {
  //...
  externals: [
    {
      // String
      react: 'react',
      // Object
      lodash : {
        commonjs: 'lodash',
        amd: 'lodash',
        root: '_' // indicates global variable
      },
      // Array
      subtract: ['./math', 'subtract']
    },
    // Function
    function(context, request, callback) {
      if (/^yourregex$/.test(request)){
        return callback(null, 'commonjs ' + request);
      }
      callback();
    },
    // Regex
    /^(jquery|\$)$/i
  ]
};
```

For more information on how to use this configuration, please refer to the article on how to author a library.

Performance

These options allows you to control how webpack notifies you of assets and entry points 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

`string = 'warning': 'error' | 'warning'  boolean: false`

Turns hints on/off. In addition, tells webpack to throw either an error or a warning when hints are found.

Given an asset is created that is over 250kb:

```
module.exports = {
  //...
  performance: {
    hints: false
  }
};
```

No hint warnings or errors are shown.

```
module.exports = {
  //...
  performance: {
    hints: 'warning'
  }
};
```

A warning will be displayed notifying you of a large asset. We recommend something like this for development environments.

```
module.exports = {
  //...
  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

```
number = 250000
```

An entry point 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 entry point size in bytes.

```
module.exports = {
  //...
  performance: {
    maxEntrypointSize: 400000
  }
};
```

performance.maxAssetSize

```
number = 250000
```

An asset is any emitted file from webpack. This option controls when webpack emits a performance hint based on individual asset size in bytes.

```
module.exports = {
  //...
  performance: {
    maxAssetSize: 100000
  }
};
```

performance.assetFilter

```
function(assetFilename) => boolean
```

This property allows webpack to control what files are used to calculate performance hints. The default function is:

```
function assetFilter(assetFilename) {
  return !(/\.map$/.test(assetFilename));
}
```

You can override this property by passing your own function in:

```
module.exports = {
  //...
```

```
  performance: {
    assetFilter: function(assetFilename) {
      return assetFilename.endsWith('.js');
    }
  }
```

The example above will only give you performance hints based on `.js` files.

Node

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

`boolean = false  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.

> *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:

```
module.exports = {
  //...
```

```
  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 = false  string: 'mock'`

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 = true  string: 'mock'`

node.global

`boolean = true`

See the source for the exact behavior of this object.

node.__filename

`string = 'mock'  boolean`

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

```
string = 'mock'  boolean
```

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 = true  string: 'mock'
```

node.setImmediate

```
boolean = true  string: 'mock' | 'empty'
```

Other node core libraries

```
boolean  string: 'mock' | 'empty'
```

> *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.

To import a built-in module, use `__non_webpack_require__`, i.e. `__non_webpack_require__('modulename')` instead of `require('modulename')`.

Example:

```
module.exports = {
  //...
  node: {
    dns: 'mock',
    fs: 'empty',
    path: true,
    url: false
  }
};
```

Stats

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.

For webpack-dev-server, this property needs to be in the `devServer` object.

For webpack-dev-middleware, this property needs to be in the webpack-dev-middleware's `options` object.

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:

```
module.exports = {
  //...
  stats: 'errors-only'
};
```

Preset	`'errors-only'`
Alternative	*none*
Description	
Only output when errors happen	

Preset	`'errors-warnings'`
Alternative	*none*
Description	
Only output errors and warnings happen	

Preset	`'minimal'`
Alternative	*none*
Description	
Only output when errors or new compilation happen	

Preset	`'none'`
Alternative	`false`
Description	
Output nothing	

Preset	`'normal'`
Alternative	`true`
Description	
Standard output	

Preset	`'verbose'`
Alternative	*none*
Description	
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.all`

A fallback value for stats options when an option is not defined. It has precedence over local webpack defaults.

```
module.exports = {
  //...
  stats: {
    all: undefined
  }
};
```

stats.assets

`boolean = true`

Tells `stats` whether to show the asset information. Set `stats.assets` to `false` to hide it.

```
module.exports = {
  //...
  stats: {
    assets: false
  }
};
```

stats.assetsSort

`string = 'id'`

Tells `stats` to sort the assets by a given field. All of the sorting fields are allowed to be used as values for `stats.assetsSort`. Use `!` prefix in the value to reverse the sort order by a given field.

```
module.exports = {
  //...
  stats: {
    assetsSort: '!size'
  }
};
```

stats.builtAt

`boolean = true`

Tells `stats` whether to add the build date and the build time information. Set `stats.builtAt` to `false` to hide it.

```
module.exports = {
  //...
  stats: {
    builtAt: false
  }
};
```

stats.cached

`boolean = true`

Tells `stats` whether to add information about the cached modules (not the ones that were built).

```
module.exports = {
  //...
  stats: {
    cached: false
  }
};
```

stats.cachedAssets

`boolean = true`

Tells `stats` whether to add information about the cached assets. Setting `stats.cachedAssets` to `false` will tell `stats` to only show the emitted files (not the ones that were built).

```
module.exports = {
  //...
  stats: {
    cachedAssets: false
  }
};
```

stats.children

`boolean = true`

Tells `stats` whether to add information about the children.

```
module.exports = {
  //...
  stats: {
```

```
    children: false
  }
```

stats.chunks

`boolean = true`

Tells `stats` whether to add information about the chunk. Setting `stats.chunks` to `false` results in a less verbose output.

```
module.exports = {
  //...
  stats: {
    chunks: false
  }
};
```

stats.chunkGroups

`boolean = true`

Tells `stats` whether to add information about the `namedChunkGroups`.

```
module.exports = {
  //...
  stats: {
    chunkGroups: false
  }
};
```

stats.chunkModules

`boolean = true`

Tells `stats` whether to add information about the built modules to information about the chunk.

```
module.exports = {
  //...
  stats: {
    chunkModules: false
  }
};
```

stats.chunkOrigins

`boolean = true`

Tells `stats` whether to add information about the origins of chunks and chunk merging.

```
module.exports = {
  //...
  stats: {
    chunkOrigins: false
  }
};
```

stats.chunksSort

`string = 'id'`

Tells `stats` to sort the chunks by a given field. All of the sorting fields are allowed to be used as values for `stats.chunksSort`. Use `!` prefix in the value to reverse the sort order by a given field.

```
module.exports = {
  //...
  stats: {
    chunksSort: 'name'
  }
};
```

stats.context

`string = '../src/'`

Sets the context directory for shortening the request information.

```
module.exports = {
  //...
  stats: {
    context: '../src/components/'
  }
};
```

stats.colors

`boolean = false  object`

Tells `stats` whether to output in the different colors.

```
module.exports = {
  //...
  stats: {
    colors: true
  }
};
```

It is also available as a CLI flag:

```
webpack-cli --colors
```

You can specify your own terminal output colors using ANSI escape sequences

```
module.exports = {
  //...
  colors: {
    green: '\u001b[32m',
  },
};
```

stats.depth

`boolean = false`

Tells `stats` whether to display the distance from the entry point for each module.

```
module.exports = {
  //...
  stats: {
    depth: true
  }
};
```

stats.entrypoints

`boolean = true`

Tells `stats` whether to display the entry points with the corresponding bundles.

```
module.exports = {
  //...
  stats: {
```

```
    entrypoints: false
  }
```

stats.env

`boolean = false`

Tells `stats` whether to display the `--env` information.

```
module.exports = {
  //...
  stats: {
    env: true
  }
};
```

stats.errors

`boolean = true`

Tells `stats` whether to display the errors.

```
module.exports = {
  //...
  stats: {
    errors: false
  }
};
```

stats.errorDetails

`boolean = true`

Tells `stats` whether to add the details to the errors.

```
module.exports = {
  //...
  stats: {
    errorDetails: false
  }
};
```

stats.excludeAssets

`array = []: string | RegExp | function (assetName) => boolean  string  RegExp  function (assetName) => boolean`

Tells `stats` to exclude the matching assets information. This can be done with a `string` , a `RegExp` , a `function` that is getting the assets name as an argument and returns a `boolean` . `stats.excludeAssets` can be an `array` of any of the above.

```
module.exports = {
  //...
  stats: {
    excludeAssets: [
      'filter',
      /filter/,
      (assetName) => assetName.contains('moduleA')
    ]
  }
};
```

stats.excludeModules

`array = []: string | RegExp | function (assetName) => boolean  string  RegExp  function (assetName) => boolean  boolean: false`

Tells `stats` to exclude the matching modules information. This can be done with a `string` , a `RegExp` , a `function` that is getting the module's source as an argument and returns a `boolean` . `stats.excludeModules` can be an `array` of any of the above. `stats.excludeModules` 's configuration is merged with the `stats.exclude` 's configuration value.

```
module.exports = {
  //...
  stats: {
    excludeModules: [
      'filter',
      /filter/,
      (moduleSource) => true
    ]
  }
};
```

Setting `stats.excludeModules` to `false` will disable the exclude behaviour.

```
module.exports = {
  //...
```

```
  stats: {
    excludeModules: false
  }
};
```

stats.exclude

See `stats.excludeModules`.

stats.hash

`boolean = true`

Tells `stats` whether to add information about the hash of the compilation.

```
module.exports = {
  //...
  stats: {
    hash: false
  }
};
```

stats.logging

`string = 'info': 'none' | 'error' | 'warn' | 'info' | 'log' | 'verbose'  boolean`

Tells `stats` whether to add logging output.

- `'none'`, `false` - disable logging
- `'error'` - errors only
- `'warn'` - errors and warnings only
- `'info'` - errors, warnings, and info messages
- `'log'`, `true` - errors, warnings, info messages, log messages, groups, clears. Collapsed groups are displayed in a collapsed state.
- `'verbose'` - log everything except debug and trace. Collapsed groups are displayed in expanded state.

```
module.exports = {
  //...
  stats: {
```

```
    logging: 'verbose'
  }
```

stats.loggingDebug

`array = []: string | RegExp | function (name) => boolean` `string` `RegExp` `function (name) => boolean`

Tells `stats` to include the debug information of the specified loggers such as Plugins or Loaders. When `stats.logging` is set to `false`, `stats.loggingDebug` option is ignored.

```
module.exports = {
  //...
  stats: {
    loggingDebug: [
      'MyPlugin',
      /MyPlugin/,
      (name) => name.contains('MyPlugin')
    ]
  }
};
```

stats.loggingTrace

`boolean = true`

Enable stack traces in the logging output for errors, warnings and traces. Set `stats.loggingTrace` to hide the trace.

```
module.exports = {
  //...
  stats: {
    loggingTrace: false
  }
};
```

stats.maxModules

`number = 15`

Set the maximum number of modules to be shown.

```
module.exports = {
  //...
  stats: {
    maxModules: 5
  }
};
```

stats.modules

`boolean = true`

Tells `stats` whether to add information about the built modules.

```
module.exports = {
  //...
  stats: {
    modules: false
  }
};
```

stats.modulesSort

`string = 'id'`

Tells `stats` to sort the modules by a given field. All of the sorting fields are allowed to be used as values for `stats.modulesSort`. Use `!` prefix in the value to reverse the sort order by a given field.

```
module.exports = {
  //...
  stats: {
    modulesSort: 'size'
  }
};
```

stats.moduleTrace

`boolean = true`

Tells `stats` to show dependencies and the origin of warnings/errors. `stats.moduleTrace` is available since webpack 2.5.0.

```
module.exports = {
  //...
```

```
  stats: {
    moduleTrace: false
  }
};
```

stats.outputPath

`boolean = true`

Tells `stats` to show the `outputPath`.

```
module.exports = {
  //...
  stats: {
    outputPath: false
  }
};
```

stats.performance

`boolean = true`

Tells `stats` to show performance hint when the file size exceeds `performance.maxAssetSize`.

```
module.exports = {
  //...
  stats: {
    performance: false
  }
};
```

stats.providedExports

`boolean = false`

Tells `stats` to show the exports of the modules.

```
module.exports = {
  //...
  stats: {
    providedExports: true
  }
};
```

stats.publicPath

`boolean = true`

Tells `stats` to show the `publicPath`.

```
module.exports = {
  //...
  stats: {
    publicPath: false
  }
};
```

stats.reasons

`boolean = true`

Tells `stats` to add information about the reasons of why modules are included.

```
module.exports = {
  //...
  stats: {
    reasons: false
  }
};
```

stats.source

`boolean = false`

Tells `stats` to add the source code of modules.

```
module.exports = {
  //...
  stats: {
    source: true
  }
};
```

stats.timings

`boolean = true`

Tells `stats` to add the timing information.

```
module.exports = {
  //...
  stats: {
    timings: false
  }
};
```

stats.usedExports

`boolean = false`

Tells `stats` whether to show which exports of a module are used.

```
module.exports = {
  //...
  stats: {
    usedExports: true
  }
};
```

stats.version

`boolean = true`

Tells `stats` to add information about the webpack version used.

```
module.exports = {
  //...
  stats: {
    version: false
  }
};
```

stats.warnings

`boolean = true`

Tells `stats` to add warnings.

```
module.exports = {
  //...
```

```
  stats: {
    warnings: false
  }
}
```

stats.warningsFilter

`array = []: string | RegExp | function (warning) => boolean` `string` `RegExp` `function (warning) => boolean`

Tells `stats` to exclude the warnings that are matching given filters. This can be done with a `string`, a `RegExp`, a `function` that is getting a warning as an argument and returns a `boolean`. `stats.warningsFilter` can be an `array` of any of the above.

```
module.exports = {
  //...
  stats: {
    warningsFilter: [
      'filter',
      /filter/,
      (warning) => true
    ]
  }
};
```

Sorting fields

For `assetsSort`, `chunksSort` and `modulesSort` there are several possible fields that you can sort items by:

- `'id'` is the item's id;
- `'name'` - a item's name that was assigned to it upon importing;
- `'size'` - a size of item in bytes;
- `'chunks'` - what chunks the item originates from (for example, if there are multiple subchunks for one chunk - the subchunks will be grouped together according to their main chunk);
- `'errors'` - amount of errors in items;
- `'warnings'` - amount of warnings in items;
- `'failed'` - whether the item has failed compilation;
- `'cacheable'` - whether the item is cacheable;

- `'built'` - whether the asset has been built;
- `'prefetched'` - whether the asset will be prefetched;
- `'optional'` - whether the asset is optional;
- `'identifier'` - identifier of the item;
- `'index'` - item's processing index;
- `'index2'`
- `'profile'`
- `'issuer'` - an identifier of the issuer;
- `'issuerId'` - an id of the issuer;
- `'issuerName'` - a name of the issuer;
- `'issuerPath'` - a full issuer object. There's no real need to sort by this field;

Extending stats behaviours

If you want to use one of the pre-defined behaviours e.g. `'minimal'` but still override one or more of the rules, see the source code. You would want to copy the configuration options from `case 'minimal': ...` and add your additional rules while providing an object to `stats`.

webpack.config.js

```
module.exports = {
  //..
  stats: {
    // copied from `'minimal'`
    all: false,
    modules: true,
    maxModules: 0,
    errors: true,
    warnings: true,
    // our additional options
    moduleTrace: true,
    errorDetails: true
  }
};
```

Other Options

These are the remaining configuration options supported by webpack.

> *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  boolean: false`

Set the value of `require.amd` or `define.amd`. Setting `amd` to `false` will disable webpack's AMD support.

webpack.config.js

```
module.exports = {
  //...
  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 = false`

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:

webpack.config.js

```
module.exports = {
  //...
  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 will be automatically enabled by default while in watch mode and webpack is set to mode `development`. To enable caching manually set it to `true`:

webpack.config.js

```
module.exports = {
  //...
  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:

webpack.config.js

```
let SharedCache = {};

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

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 = 100`

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.

Use the StatsPlugin for more control over the generated profile.

Combine with `parallelism: 1` for better results.

recordsPath

`string`

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:

webpack.config.js

```
module.exports = {
  //...
  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.

> *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.*

> *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

`string`

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

`string`

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:

webpack.config.js

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

name

`string`

Name of the configuration. Used when loading multiple configurations.

webpack.config.js

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

infrastructureLogging

Options for infrastructure level logging.

```
object = {}
```

infrastructureLogging.level

```
string
```

Enable infrastructure logging output. Similar to `stats.logging` option but for infrastructure. No default value is given.

Possible values:

- `'none'` - disable logging
- `'error'` - errors only
- `'warn'` - errors and warnings only
- `'info'` - errors, warnings, and info messages
- `'log'` - errors, warnings, info messages, log messages, groups, clears. Collapsed groups are displayed in a collapsed state.
- `'verbose'` - log everything except debug and trace. Collapsed groups are displayed in expanded state.

webpack.config.js

```
module.exports = {
  //...
  infrastructureLogging: {
    level: 'info'
  }
};
```

infrastructureLogging.debug

`string` `RegExp` `function(name) => boolean` `[string, RegExp, function(name) => boolean]`

Enable debug information of specified loggers such as plugins or loaders. Similar to `stats.loggingDebug` option but for infrastructure. No default value is given.

webpack.config.js

```
module.exports = {
  //...
  infrastructureLogging: {
    level: 'info',
    debug: [
      'MyPlugin',
      /MyPlugin/,
      (name) => name.contains('MyPlugin')
    ]
  }
};
```