Table of Contents

CONCEPTS

GUIDES

DOCUMENTATION

PLUGINS

API

DEVELOPMENT

LOADERS

PLUGINSAPI

Introduction

webpack is a *module bundler* for modern JavaScript applications. It is incredibly configurable, however, there are **Four Core Concepts** we feel you should understand before you get started!

As part of your webpack learning journey, we wrote this document aimed to give you a **high-level** overview of these concepts, while still providing links to concept specific use-cases.

Entry

webpack creates a graph of all of your application's dependencies. The starting point of this graph is known as an *entry point*. The *entry point* tells webpack *where to start* and follows the graph of dependencies to know *what to bundle*. You can think of your application's *entry point* as the **contextual root** or **the first file to kick off your app**.

In webpack we define *entry points* using the `entry` property in our webpack configuration object.

The simplest example is seen below:

webpack.config.js

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

There are multiple ways to declare your `entry` property that are specific to your application's needs.

Learn more!

Output

Once you've bundled all of your assets together, we still need to tell webpack **where** to bundle our application. The webpack `output` property describes to webpack **how to treat bundled code**.

webpack.config.js

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

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

In the example above, through the `output.filename` and `output.path` properties we are describing to webpack the name of our bundle, and where we want it to be emitted to.

T> You may see the term **emitted** or **emit** used throughout our documentation and plugin API. This is a fancy term for "produced or discharged".

The `output` property has many more configurable features, but let's spend some time understanding some of the most common use cases for the `output` property.

Learn more!

Loaders

The goal is to have all of the assets in your project to be **webpack's** concern and not the browser's. (This doesn't mean that they all have to be bundled together). webpack treats every file (.css, .html, .scss, .jpg, etc.) as a module. However, webpack **only understands JavaScript**.

Loaders in webpack *transform these files into modules* as they are added to your dependency graph.

At a high level, they have two purposes in your webpack config.

1. Identify what files should be transformed by a certain loader. (`test` property)
2. Transform that file so that it can be added to your dependency graph (and eventually your bundle). (`use` property)

webpack.config.js

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

const config = {
  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'}
    ]
  }
};

module.exports = config;
```

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

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

W> It is important to remember when defining rules in your webpack config, you are defining them under `module.rules`, and not `rules`. But webpack will yell at you when doing this incorrectly.

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

Learn more!

Plugins

Since Loaders only execute transforms on a per-file basis, `plugins` are most commonly used (but not limited to) performing actions and custom functionality on "compilations" or "chunks" of your bundled modules (and so much more). The webpack Plugin system is extremely powerful and customizable.

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

webpack.config.js

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

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

module.exports = config;
```

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

Using plugins in your webpack config is straight-forward, however there are many use-cases that are worth discussing further.

Learn more!

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

Single Entry (Shorthand) Syntax

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

webpack.config.js

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

module.exports = config;
```

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

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

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

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

Object Syntax

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

webpack.config.js

```
const config = {
```

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

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

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

Scenarios

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

Separate App and Vendor Entries

webpack.config.js

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

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

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

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

Multi Page Application

webpack.config.js

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

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

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

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

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

Options affecting the output of the compilation. `output` options tell webpack how to write the compiled files to disk. Note, that while there can be multiple `entry` points, only one `output` configuration is specified.

If you use any hashing (`[hash]` or `[chunkhash]`), make sure to have a consistent ordering of modules. Use the `OccurrenceOrderPlugin` or `recordsPath` .

Usage

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

Your preferred `filename` of the compiled file: `// main.js || bundle.js || index.js`

An `output.path` as an **absolute path** for what directory you prefer it to go in once bundled.

webpack.config.js

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

module.exports = config;
```

Options

The following is a list of values you can pass to the `output` property.

`output.chunkFilename`

The filename of non-entry chunks as a relative path inside the `output.path` directory.

`[id]` is replaced by the id of the chunk.

`[name]` is replaced by the name of the chunk (or with the id when the chunk has no name).

`[hash]` is replaced by the hash of the compilation.

`[chunkhash]` is replaced by the hash of the chunk.

`output.crossOriginLoading`

This option enables cross-origin loading of chunks.

Possible values are:

`false` - Disable cross-origin loading.

`"anonymous"` - Cross-origin loading is enabled. When using `anonymous` no credentials will be sent with the request.

`"use-credentials"` - Cross-origin loading is enabled and credentials will be send with the request.

For more information on cross-origin loading see MDN

> Default: `false`
>
> see also library
>
> see also Development Tools

`output.devtoolLineToLine`

Enable line-to-line mapped mode for all/specified modules. Line-to-line mapped mode uses a simple SourceMap where each line of the generated source is mapped to the same line of the original source. It's a performance optimization. Only use it if your performance needs to be better and you are sure that input lines match which generated lines.

`true` enables it for all modules (not recommended)

An object `{test, include, exclude}` similar to `module.loaders` enables it for specific files.

> Default: `false`

`output.filename`

Specifies the name of each output file on disk. You must **not** specify an absolute path here! The `output.path` option determines the location on disk the files are written. `filename` is used solely for naming the individual files.

single entry

```
{
  entry: './src/app.js',
  output: {
    filename: 'bundle.js',
    path: __dirname + '/build'
  }
}

// writes to disk: ./build/bundle.js
```

multiple entries

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

`[name]` is replaced by the name of the chunk.

`[hash]` is replaced by the hash of the compilation.

`[chunkhash]` is replaced by the hash of the chunk.

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

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

`output.hotUpdateChunkFilename`

The filename of the Hot Update Chunks. They are inside the `output.path` directory.

`[id]` is replaced by the id of the chunk.

`[hash]` is replaced by the hash of the compilation. (The last hash stored in the records)

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

`output.hotUpdateFunction`

The JSONP function used by webpack for async loading of hot update chunks.

> Default: `"webpackHotUpdate"`

`output.hotUpdateMainFilename`

The filename of the Hot Update Main File. It is inside the `output.path` directory.

`[hash]` is replaced by the hash of the compilation. (The last hash stored in the records)

> Default: `"[hash].hot-update.json"`

`output.jsonpFunction`

The JSONP function used by webpack for async loading of chunks.

A shorter function may reduce the file size a bit. Use a different identifier when having multiple webpack instances on a single page.

> Default: `"webpackJsonp"`

`output.library`

If set, export the bundle as library. `output.library` is the name.

Use this if you are writing a library and want to publish it as single file.

`output.libraryTarget`

Which format to export the library:

`"var"` - Export by setting a variable: `var Library = xxx` (default)

`"this"` - Export by setting a property of `this`: `this["Library"] = xxx`

`"commonjs"` - Export by setting a property of `exports`: `exports["Library"] = xxx`

`"commonjs2"` - Export by setting `module.exports`: `module.exports = xxx`

`"amd"` - Export to AMD (optionally named - set the name via the library option)

`"umd"` - Export to AMD, CommonJS2 or as property in root

> Default: `"var"`

If `output.library` is not set, but `output.libraryTarget` is set to a value other than `var`, every property of the exported object is copied (Except `amd`, `commonjs2` and `umd`).

`output.path`

The output directory as an **absolute path** (required).

`[hash]` is replaced by the hash of the compilation.

config.js

```
output: {
    path: "/home/proj/public/assets",
    publicPath: "/assets/"
}
```

index.html

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

And a more complicated example of using a CDN and hashes for assets.

config.js

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

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

```
 __webpack_public_path__ = myRuntimePublicPath

// rest of your application entry
```

`output.sourceMapFilename`

The filename of the SourceMaps for the JavaScript files. They are inside the `output.path` directory.

`[file]` is replaced by the filename of the JavaScript file.

`[id]` is replaced by the id of the chunk.

`[hash]` is replaced by the hash of the compilation.

> Default: `"[file].map"`

Loaders are transformations that are applied on a resource file of your application. They are functions (running in Node.js) that take the source of a resource file as the parameter and return the new source.

Example

For example, you can use loaders to tell webpack to load a CSS file or to convert TypeScript to JavaScript. Firstly, install the corresponding loaders:

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

Secondly, configure in your `webpack.config.js` that for every `.css` file the `css-loader` should be used and analogously for `.ts` files and the `ts-loader`:

webpack.config.js

```
module.exports = {
  module: {
    rules: [
      {test: /\.css$/, use: ['css-loader'](/loaders/css-loader)},
      {test: /\.ts$/, use: ['ts-loader'](https://github.com/TypeStrong/ts-loader)}
    ]
  }
};
```

Note that according to the configuration options, the following specifications define the identical loader usage:

```
{test: /\.css$/, [loader](/configuration/module#rule-loader): 'css-loader'}
// or equivalently
{test: /\.css$/, [use](/configuration/module#rule-use): 'css-loader'}
// or equivalently
{test: /\.css$/, [use](/configuration/module#rule-use): {
  loader: 'css-loader',
  options: {}
}}
```

Configuration

There are three ways to use loaders in your application:

- via configuration
- explicit in the `require` statement
- via CLI

Via `webpack.config.js`

`module.rules` allows you to specify several loaders within your webpack configuration. This is a concise way to display loaders, and helps to have clean code as well as you have a full overview of each respective loader.

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

Via `require`

It's possible to specify the loaders in the `require` statement (or `define`, `require.ensure`, etc.). Separate loaders from the resource with `!`. Each part is resolved relative to the current directory.

```
require('style-loader!css-loader?modules!./styles.css');
```

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

Options can be passed with a query parameter, just like on the web (`?key=value&foo=bar`). It's also possible to use a JSON object (`?{"key":"value","foo":"bar"}`).

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

Via CLI

Optionally, you could also use loaders through the CLI:

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

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

Loader Features

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

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

Resolving Loaders

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

How to write a loader? A loader module is expected to export a function and to be written in Node.js compatible JavaScript. In the common case you manage loaders with npm, but you can also have loaders as files in your app.

By convention, loaders are usually named as `xxx-loader`, where `xxx` is the context name. For example, `json-loader`.

The loader name convention and precedence search order is defined by `resolveLoader.moduleTemplates` within the webpack configuration API.

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

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

Anatomy

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

ConsoleLogOnBuildWebpackPlugin.js

```
function ConsoleLogOnBuildWebpackPlugin() {

};

ConsoleLogOnBuildWebpackPlugin.prototype.apply = function(compiler) {
  compiler.plugin('run', function(compiler, callback) {
    console.log("The webpack build process is starting!!!");

    callback();
  });
};
```

T> As a clever JavaScript developer you may remember the `Function.prototype.apply` method. Because of this method you can pass any function as plugin (`this` will point to the `compiler`). You can use this style to inline custom plugins in your configuration.

Usage

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

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

Configuration

webpack.config.js

```
const HtmlWebpackPlugin = require('html-webpack-plugin'); //installed via npm
```

```
const webpack = require('webpack'); //to access built-in plugins
const path = require('path');

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

module.exports = config;
```

Node API

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

some-node-script.js

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

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

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

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

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

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

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

Use these features when appropriate.

You should NOT use the following things. Technically you could use them, but it's **not recommended**:

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

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

The Simplest Configuration

webpack.config.js

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

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

Multiple Targets

webpack.config.js

```
var path = require('path');
var webpack = require('webpack');
var webpackMerge = require('webpack-merge');

var baseConfig = {
  target: 'async-node',
  entry: {
    entry: './entry.js'
  },
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: '[name].js'
  },
  plugins: [
    new webpack.optimize.CommonsChunkPlugin({
      name: 'inline',
      filename: 'inline.js',
      minChunks: Infinity
    }),
    new webpack.optimize.AggressiveSplittingPlugin({
        minSize: 5000,
        maxSize: 10000
    }),
  ]
};

let targets = ['web', 'webworker', 'node', 'async-node', 'node-webkit', 'electron-
main'].map((target) => {
  let base = webpackMerge(baseConfig, {
    target: target,
    output: {
      path: path.resolve(__dirname, 'dist/' + target),
      filename: '[name].' + target + '.js'
    }
  });
  return base;
});

module.exports = targets;
```

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

Using TypeScript

In the example below we use TypeScript to create a class which the angular-cli uses to generate configs.

webpack.config.ts

```
import * as webpackMerge from 'webpack-merge';
import { CliConfig } from './config';
import {
  getWebpackCommonConfig,
  getWebpackDevConfigPartial,
  getWebpackProdConfigPartial,
  getWebpackMobileConfigPartial,
  getWebpackMobileProdConfigPartial
} from './';

export class NgCliWebpackConfig {
  // TODO: When webpack2 types are finished let's replace all these any types
  // so this is more maintainable in the future for devs
  public config: any;
  private webpackDevConfigPartial: any;
  private webpackProdConfigPartial: any;
  private webpackBaseConfig: any;
  private webpackMobileConfigPartial: any;
  private webpackMobileProdConfigPartial: any;

  constructor(public ngCliProject: any, public target: string, public environment:
 string, outputDir?: string) {
    const config: CliConfig = CliConfig.fromProject();
    const appConfig = config.config.apps[0];

    appConfig.outDir = outputDir || appConfig.outDir;

    this.webpackBaseConfig = getWebpackCommonConfig(this.ngCliProject.root, enviro
nment, appConfig);
    this.webpackDevConfigPartial = getWebpackDevConfigPartial(this.ngCliProject.ro
ot, appConfig);
    this.webpackProdConfigPartial = getWebpackProdConfigPartial(this.ngCliProject.
root, appConfig);

    if (appConfig.mobile){
      this.webpackMobileConfigPartial = getWebpackMobileConfigPartial(this.ngCliPr
oject.root, appConfig);
      this.webpackMobileProdConfigPartial = getWebpackMobileProdConfigPartial(this
.ngCliProject.root, appConfig);
      this.webpackBaseConfig = webpackMerge(this.webpackBaseConfig, this.webpackMo
bileConfigPartial);
      this.webpackProdConfigPartial = webpackMerge(this.webpackProdConfigPartial,
this.webpackMobileProdConfigPartial);
    }
```

```
    this.generateConfig();
  }

  generateConfig(): void {
    switch (this.target) {
      case "development":
        this.config = webpackMerge(this.webpackBaseConfig, this.webpackDevConfigPa
rtial);
        break;
      case "production":
        this.config = webpackMerge(this.webpackBaseConfig, this.webpackProdConfigP
artial);
        break;
      default:
        throw new Error("Invalid build target. Only 'development' and 'production'
 are available.");
        break;
    }
  }
}
```

Using 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)

```
import h from 'jsxobj';

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

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

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

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

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

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

What is a webpack Module

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

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

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

Supported Module Types

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

- CoffeeScript
- TypeScript
- ESNext (Babel)
- Sass

- Less
- Stylus

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

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

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

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

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

Resolving rules in webpack

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

Absolute paths

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

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

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

Relative paths

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

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

Module paths

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

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

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

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

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

- If the folder contains a `package.json` file, then fields specified in `resolve.mainFields` configuration option are looked up in order, and the first such field in `package.json` determines the file path.
- If there is no `package.json` or if the main fields do not return a valid path, file names specified in the `resolve.mainFiles` configuration option are looked for in order, to see if a matching filename exists in the imported/required directory .
- The file extension is then resolved in a similar way using the `resolve.extensions` option.

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

Resolving Loaders

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

Caching

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

Look at Resolve API to know more on the configuration options mentioned above.

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

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

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

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

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

Usage

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

webpack.config.js

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

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

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

?>Further expansion for other popular target values

Multiple Targets

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

webpack.config.js

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

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

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

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

Resources

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

Bundle Output Comparison

compare-webpack-target-bundles: A great resource for testing and viewing different webpack *targets*. Also great for bug reporting.

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

Hot Module Replacement (HMR) exchanges, adds, or removes modules while an application is running without a page reload. This allows you to speed up development time by updating individual modules when they are changed without refreshing the page.

How Does It Work?

From The App View

1. The app code asks the HMR runtime to check for updates.
2. The HMR runtime downloads the updates (asynchronously) and tells the app code that an update is available.
3. The app code then asks the HMR runtime to apply the updates.
4. The HMR runtime applies the update (synchronously).

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

From The Compiler (webpack) View

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

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

The manifest contains the new compilation hash and a list of all update chunks.

Each update chunk contains code for all updated modules in the respective chunk (or a flag indicating that the module was removed).

The compiler makes sure that module IDs and chunk IDs are consistent between these builds. It typically stores these IDs in memory (for example, when using webpack-dev-server), but it's also possible to store them in a JSON file.

From The Module View

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

Similarly, when implementing the HMR interface in a module, you can describe what should happen when the module is updated. However, in most cases, it's not mandatory to write HMR code in every module. If a module has no HMR handlers, the update bubbles up. This means that a single handler can handle an update to a complete module tree. If a single module in this tree is updated, the complete module tree is reloaded (only reloaded, not transferred).

From The HMR Runtime View (Technical)

For the module system runtime, additional code is emitted to track module `parents` and `children`.

On the management side, the runtime supports two methods: `check` and `apply`.

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

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

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

What can I do with it?

You can use it in development as a LiveReload replacement. webpack-dev-server supports a hot mode in which it tries to update with HMR before trying to reload the whole page. See how to implement HMR with React as an example.

Some loaders already generate modules that are hot-updatable. For example, the `style-loader` can swap out a page's stylesheets. For modules like this, you don't need to do anything special.

webpack's power lies in its customizability, and there are *many* ways of configuring HMR depending on the needs of a particular project.

This section contains guides for understanding and mastering the wide variety of tools and features that webpack offers. The first is a simple guide that takes you through installation.

The guides get progressively more advanced as you go on. Most serve as a starting point, and once completed you should feel more comfortable diving into the actual documentation.

webpack is a tool to build JavaScript modules in your application. To start using `webpack` from its cli or api, follow the Installation instructions. webpack simplifies your workflow by quickly constructing a dependency graph of your application and bundling them in the right order. webpack can be configured to customise optimisations to your code, to split vendor/css/js code for production, run a development server that hot-reloads your code without page refresh and many such cool features. Learn more about why you should use webpack.

Creating a bundle

Create a demo directory to try out webpack. Install webpack.

```
mkdir webpack-demo && cd webpack-demo
npm init -y
npm install --save-dev webpack
```

```
./node_modules/.bin/webpack --help # Shows a list of valid cli commands
.\node_modules\.bin\webpack --help # For windows users
webpack --help # If you installed webpack globally
```

Now create a subdirectory `app` with an `index.js` file.

app/index.js

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

  /* lodash is required for the next line to work */
  element.innerHTML = _.join(['Hello','webpack'], ' ');

  return element;
}

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

To run this piece of code, one usually has the below HTML

index.html

```
<html>
  <head>
    <title>webpack 2 demo</title>
```

```
    <script src="https://unpkg.com/lodash@4.16.6"></script>
  </head>
  <body>
    <script src="app/index.js"></script>
  </body>
</html>
```

In this example, there are implicit dependencies between the `<script>` tags.

`index.js` depends on `lodash` being included in the page before it runs. It is implicit because `index.js` never declared a need for `lodash`; it just assumes that a global variable `_` exists.

There are problems with managing JavaScript projects this way:

- If a dependency is missing, or is included in the wrong order, the application will not function at all.
- If a dependency is included but is not used, then there is a lot of unnecessary code that the browser has to download.

To bundle the `lodash` dependency with `index.js`, we need to first install `lodash`

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

and then import it.

app/index.js

```
+ import _ from 'lodash';

function component () {
  ...
```

We also need to change `index.html` to expect a single bundled js file.

```
<html>
  <head>
    <title>webpack 2 demo</title>
-   <script src="https://unpkg.com/lodash@4.16.6"></script>
  </head>
  <body>
-   <script src="app/index.js"></script>
+   <script src="dist/bundle.js"></script>
  </body>
</html>
```

Here, `index.js` explicitly requires `lodash` to be present, and binds it as `_` (no global scope pollution).

By stating what dependencies a module needs, webpack can use this information to build a dependency graph. It then uses the graph to generate an optimized bundle where scripts will be executed in the correct order. Also unused dependencies will not be included in the bundle.

Now run `webpack` on this folder with `index.js` as the entry file and `bundle.js` as the output file in which all code required for the page is bundled.

```
./node_modules/.bin/webpack app/index.js dist/bundle.js

Hash: ff6c1d39b26f89b3b7bb
Version: webpack 2.2.0
Time: 385ms
    Asset    Size  Chunks                    Chunk Names
bundle.js  544 kB       0  [emitted]  [big]  main
   [0] ./~/lodash/lodash.js 540 kB {0} [built]
   [1] (webpack)/buildin/global.js 509 bytes {0} [built]
   [2] (webpack)/buildin/module.js 517 bytes {0} [built]
   [3] ./app/index.js 278 bytes {0} [built]
```

T> Output may vary. If the build is successful then you are good to go.

Open `index.html` in your browser to see the result of a successful bundle. You should see a page with the following text: 'Hello webpack'.

Using ES2015 modules with webpack

Noticed the use of ES2015 module import (alias ES2015, *harmony*) in `app/index.js`? Although `import` / `export` statements are not supported in browsers (yet), using them is fine since webpack will replace those instructions with an ES5 compatible wrapper code. Inspect `dist/bundle.js` to convince yourself.

Note that webpack will not touch your code other than `import` / `export`. In case you are using other ES2015 features, make sure to use a transpiler such as Babel or Bublé.

Using webpack with a config

For a more complex configuration, we can use a configuration file that webpack can reference to bundle your code. After you create a `webpack.config.js` file, you can represent the CLI command above with the following config settings.

webpack.config.js

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

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

This file can be run by webpack as follows.

```
webpack --config webpack.config.js

Hash: ff6c1d39b26f89b3b7bb
Version: webpack 2.2.0
Time: 390ms
    Asset    Size  Chunks                    Chunk Names
bundle.js  544 kB       0  [emitted]  [big]  main
   [0] ./~/lodash/lodash.js 540 kB {0} [built]
   [1] (webpack)/buildin/global.js 509 bytes {0} [built]
   [2] (webpack)/buildin/module.js 517 bytes {0} [built]
   [3] ./app/index.js 278 bytes {0} [built]
```

T> If a `webpack.config.js` is present, `webpack` command picks it up by default.

T> If you created a successful `dist/bundle.js` file using the 'Creating a bundle' section, delete the `dist` subdirectory to validate output from your `webpack.config.js` file settings.

The config file allows for all the flexibility in using webpack. We can add loader rules, plugins, resolve options and many other enhancements to our bundles using this configuration file.

Using webpack with npm

Given it's not particularly fun to run webpack from the CLI this way, we can set up a little shortcut. Adjust *package.json* like this:

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

You can now achieve the same as above by using `npm run build` command. npm picks up the scripts through it and patches the environment temporarily so that it contains the bin commands. You will see this convention in a lot of projects out there.

T> You can pass custom parameters to webpack by adding two dashes to the `npm run build` command, e.g. `npm run build -- --colors`.

Conclusion

Now that you have a basic build together, you should dig into the basic concepts and configuration of webpack to better understand its design. Also check out the guides to learn how to approach common problems. The API section digs into the lower level features.

Pre-requisites

Before getting started, make sure you have a fresh version of Node.js installed. The current LTS is an ideal starting point. You may run into a variety of issues with the older versions as they may be missing functionality webpack or related packages might need.

The next section tells you how to install webpack locally in a project.

Local Installation

```
npm install webpack --save-dev

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

If you are using npm scripts in your project, npm will try to look for webpack installation in your local modules for which this installation technique is useful.

```
"scripts": {
    "start": "webpack --config mywebpack.config.js"
}
```

This is standard and recommended practice.

T> To run the local installation of webpack you can access its bin version as `node_modules/.bin/webpack`

Global Installation

W> Note that a global webpack installation is not a recommended practice. This locks you down to a specific version of webpack and might fail in projects that use a different version.

```
npm install webpack -g
```

The `webpack` command is now available globally.

Bleeding Edge

If you are enthusiastic about using the latest that webpack has to offer (beware - may be unstable), you can install directly from the webpack repository using

```
npm install webpack/webpack#<tagname/branchname>
```

`resolve.root`, `resolve.fallback`, `resolve.modulesDirectories`

These options were replaced by a single option `resolve.modules`. See resolving for more usage.

```
  resolve: {
-   root: path.join(__dirname, "src")
+   modules: [
+     path.join(__dirname, "src"),
+     "node_modules"
+   ]
  }
```

`resolve.extensions`

This option no longer requires passing an empty string. This behavior was moved to `resolve.enforceExtension`. See resolving for more usage.

`resolve.*`

More stuff was changed here. Not listed in detail as it's not commonly used. See resolving for details.

`module.loaders` is now `module.rules`

The old loader configuration was superseded by a more powerful rules system, which allows configuration of loaders and more. For compatibility reasons, the old `module.loaders` syntax is still valid and the old names are parsed. The new naming conventions are easier to understand and are a good reason to upgrade the configuration to using `module.rules`.

```
  module: {
-   loaders: [
+   rules: [
      {
        test: /\.css$/,
-       loaders: [
```

```
+       use: [
          {
            loader: "style-loader"
          },
          {
            loader: "css-loader",
-           query: {
+           options: {
              modules: true
            }
          }
        ]
      },
      {
        test: /\.jsx$/,
        loader: "babel-loader", // Do not use "use" here
        options: {
          // ...
        }
      }
    ]
  }
```

Chaining loaders

Like in webpack v1, loaders can be chained to pass results from loader to loader. Using the rule.use configuration option, `use` can be set to a list of loaders. In webpack v1, loaders were commonly chained with `!`. This style is only supported using the legacy option `module.loaders`.

```
  module: {
-   loaders: {
+   rules: [{
      test: /\.less$/,
-     loader: "style-loader!css-loader!less-loader"
+     use: [
+       "style-loader",
+       "css-loader",
+       "less-loader"
+     ]
    }]
  }
```

Automatic `-loader` module name extension removed

It is not possible anymore to omit the `-loader` extension when referencing loaders:

```
  module: {
    rules: [
      {
        use: [
-         "style",
+         "style-loader",
-         "css",
+         "css-loader",
-         "less",
+         "less-loader",
        ]
      }
    ]
  }
```

You can still opt-in to the old behavior with the `resolveLoader.moduleExtensions` configuration option, but this is not recommended.

```
+ resolveLoader: {
+   moduleExtensions: ["-loader"]
+ }
```

See #2986 for the reason behind this change.

`json-loader` is not required anymore

When no loader has been configured for a JSON file, webpack will automatically try to load the JSON file with the `json-loader`.

```
  module: {
    rules: [
-     {
-       test: /\.json/,
-       loader: "json-loader"
-     }
    ]
  }
```

We decided to do this in order to iron out environment differences between webpack, node.js and browserify.

Loaders in configuration resolve relative to context

In webpack 1 configured loaders resolve relative to the matched file. Since webpack 2 configured loaders resolve relative to the `context` option.

This solves some problems with duplicate modules caused by loaders when using `npm link` or referencing modules outside of the `context`.

You may remove some hacks to work around this:

```
  module: {
    rules: [
      {
        // ...
-       loader: require.resolve("my-loader")
+       loader: "my-loader"
      }
    ]
  },
  resolveLoader: {
-   root: path.resolve(__dirname, "node_modules")
  }
```

`module.preLoaders` and `module.postLoaders` was removed

```
  module: {
-   preLoaders: [
+   rules: [
      {
        test: /\.js$/,
+       enforce: "pre",
        loader: "eslint-loader"
      }
    ]
  }
```

`UglifyJsPlugin` sourceMap

The `sourceMap` option of the `UglifyJsPlugin` now defaults to `false` instead of `true`. This means that if you are using source maps for minimized code or want correct line numbers for uglifyjs warnings, you need to set `sourceMap: true` for `UglifyJsPlugin`.

```
  devtool: "source-map",
  plugins: [
    new UglifyJsPlugin({
+     sourceMap: true
    })
  ]
```

`UglifyJsPlugin` warnings

The `compress.warnings` option of the `UglifyJsPlugin` now defaults to `false` instead of `true`. This means that if you want to see uglifyjs warnings, you need to set `compress.warnings` to `true`.

```
  devtool: "source-map",
  plugins: [
    new UglifyJsPlugin({
+     compress: {
+       warnings: true
+     }
    })
  ]
```

`UglifyJsPlugin` minimize loaders

`UglifyJsPlugin` no longer switches loaders into minimize mode. The `minimize: true` setting needs to be passed via loader options in long-term. See loader documentation for relevant options.

The minimize mode for loaders will be removed in webpack 3 or later.

To keep compatibility with old loaders, loaders can be switched to minimize mode via plugin:

```
  plugins: [
```

```
+   new webpack.LoaderOptionsPlugin({
+     minimize: true
+   })
  ]
```

`DedupePlugin` has been removed

`webpack.optimize.DedupePlugin` isn't needed anymore. Remove it from your configuration.

`BannerPlugin` - breaking change

`BannerPlugin` no longer accept two parameters but rather only a single options object.

```
  plugins: [
-    new webpack.BannerPlugin('Banner', {raw: true, entryOnly: true});
+    new webpack.BannerPlugin({banner: 'Banner', raw: true, entryOnly: true});
  ]
```

`OccurrenceOrderPlugin` is now on by default

It's no longer necessary to specify it in configuration.

```
  plugins: [
-   new webpack.optimize.OccurrenceOrderPlugin()
  ]
```

`ExtractTextWebpackPlugin` - breaking change

ExtractTextPlugin 1.0.0 does not work with webpack v2. You will need to install ExtractTextPlugin v2 explicitly.

```
npm install --save-dev extract-text-webpack-plugin@beta
```

The configuration changes for this plugin are mainly syntactical.

`ExtractTextPlugin.extract`

```
module: {
  rules: [
    {
      test: /.css$/,
-      loader: ExtractTextPlugin.extract("style-loader", "css-loader", { publicPat
h: "/dist" })
+      use: ExtractTextPlugin.extract({
+        fallback: "style-loader",
+        use: "css-loader",
+        publicPath: "/dist"
+      })
    }
  ]
}
```

`new ExtractTextPlugin({options})`

```
plugins: [
-  new ExtractTextPlugin("bundle.css", { allChunks: true, disable: false })
+  new ExtractTextPlugin({
+    filename: "bundle.css",
+    disable: false,
+    allChunks: true
+  })
]
```

Full dynamic requires now fail by default

A dependency with only an expression (i. e. `require(expr)`) will now create an empty context instead of an context of the complete directory.

Best refactor this code as it won't work with ES2015 Modules. If this is not possible you can use the `ContextReplacementPlugin` to hint the compiler to the correct resolving.

?> Link to an article about dynamic dependencies.

Using custom arguments in CLI and configuration

If you abused the CLI to pass custom arguments to the configuration like so:

`webpack --custom-stuff`

```
// webpack.config.js
var customStuff = process.argv.indexOf("--custom-stuff") >= 0;
```

```
/* ... */
module.exports = config;
```

You may notice that this is no longer allowed. The CLI is more strict now.

Instead there is an interface for passing arguments to the configuration. This should be used instead. Future tool may rely on this.

`webpack --env.customStuff`

```
module.exports = function(env) {
  var customStuff = env.customStuff;
  /* ... */
  return config;
};
```

See CLI.

`require.ensure` and AMD `require` is asynchronous

These functions are now always asynchronous instead of calling their callback sync if the chunk is already loaded.

nb `require.ensure` now depends upon native `Promise`s. If using `require.ensure` in an environment that lacks them then you will need a polyfill.

Loader configuration is through `options`

You can *no longer* configure a loader with a custom property in the `webpack.config.js`. It must be done through the `options`. The following configuration with the `ts` property is no longer valid with webpack 2:

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

```
    // does not work with webpack 2
    ts: { transpileOnly: false }
  }
```

What are `options`?

Good question. Well, strictly speaking it's 2 possible things; both ways to configure a webpack loader. Classically `options` was called `query` and was a string which could be appended to the name of the loader. Much like a query string but actually with greater powers:

```
module.exports = {
  ...
  module: {
    rules: [{
      test: /\.tsx?$/,
      loader: 'ts-loader?' + JSON.stringify({ transpileOnly: false })
    }]
  }
}
```

But it can also be a separately specified object that's supplied alongside a loader:

```
module.exports = {
  ...
  module: {
    rules: [{
      test: /\.tsx?$/,
      loader: 'ts-loader',
      options:  { transpileOnly: false }
    }]
  }
}
```

`LoaderOptionsPlugin` context

Some loaders need context information and read them from the configuration. This need to be passed via loader options in long-term. See loader documentation for relevant options.

To keep compatibility with old loaders, this information can be passed via plugin:

```
  plugins: [
+   new webpack.LoaderOptionsPlugin({
+     options: {
+       context: __dirname
+     }
+   })
  ]
```

debug

The `debug` option switched loaders to debug mode in webpack 1. This need to be passed via loader options in long-term. See loader documentation for relevant options.

The debug mode for loaders will be removed in webpack 3 or later.

To keep compatibility with old loaders, loaders can be switched to debug mode via plugin:

```
- debug: true,
  plugins: [
+   new webpack.LoaderOptionsPlugin({
+     debug: true
+   })
  ]
```

Code Splitting with ES2015

In webpack v1, you could use `require.ensure` as a method to lazily-load chunks for your application:

```
require.ensure([], function(require) {
  var foo = require("./module");
});
```

The ES2015 Loader spec defines `import()` as method to load ES2015 Modules dynamically on runtime.

webpack treats `import()` as a split-point and puts the requested module in a separate chunk.

`import()` takes the module name as argument and returns a Promise.

```
function onClick() {
  import("./module").then(module => {
    return module.default;
  }).catch(err => {
    console.log("Chunk loading failed");
  });
}
```

Good news: Failure to load a chunk can be handled now because they are `Promise` based.

Caveat: `require.ensure` allows for easy chunk naming with the optional third argument, but `import` API doesn't offer that capability yet. If you want to keep that functionality, you can continue using `require.ensure`.

```
require.ensure([], function(require) {
  var foo = require("./module");
}, "custom-chunk-name");
```

(Note on the deprecated `System.import`: webpack's use of `System.import` didn't fit the proposed spec, so it was deprecated in v2.1.0-beta.28 in favor of `import()`)

If you want to use `import` with Babel, you'll need to install/add the dynamic-import syntax plugin while it's still Stage 3 to get around the parser error. When the proposal is added to the spec this won't be necessary anymore.

Dynamic expressions

It's possible to pass a partial expression to `import()`. This is handled similar to expressions in CommonJS (webpack creates a context with all possible files).

`import()` creates a separate chunk for each possible module.

```
function route(path, query) {
  return import(`./routes/${path}/route`)
    .then(route => new route.Route(query));
}
// This creates a separate chunk for each possible route
```

Mixing ES2015 with AMD and CommonJS

As for AMD and CommonJS you can freely mix all three module types (even within the same file). webpack behaves similar to babel and node-eps in this case:

```
// CommonJS consuming ES2015 Module
var book = require("./book");

book.currentPage;
book.readPage();
book.default === "This is a book";
```

```
// ES2015 Module consuming CommonJS
import fs from "fs"; // module.exports map to default
import { readFileSync } from "fs"; // named exports are read from returned object+

typeof fs.readFileSync === "function";
typeof readFileSync === "function";
```

It is important to note that you will want to tell Babel to not parse these module symbols so webpack can use them. You can do this by setting the following in your `.babelrc` or `babel-loader` options.

.babelrc

```
{
  "presets": [
    ["es2015", { "modules": false }]
  ]
}
```

Hints

No need to change something, but opportunities

Template strings

webpack now supports template strings in expressions. This means you can start using them in webpack constructs:

```
- require("./templates/" + name);
+ require(`./templates/${name}`);
```

Configuration Promise

webpack now supports returning a `Promise` from the configuration file. This allows to do async processing in you configuration file.

webpack.config.js

```
module.exports = function() {
  return fetchLangs().then(lang => ({
    entry: "...",
    // ...
    plugins: [
      new DefinePlugin({ LANGUAGE: lang })
    ]
  }));
};
```

Advanced loader matching

webpack now supports more things to match on for loaders.

```
module: {
  rules: [
    {
      resource: /filename/, // matches "/path/filename.js"
      resourceQuery: /querystring/, // matches "/filename.js?querystring"
      issuer: /filename/, // matches "/path/something.js" if requested from "/path
/filename.js"
    }
  ]
}
```

More CLI options

There are some new CLI options for you to use:

`--define process.env.NODE_ENV="production"` See `DefinePlugin`.

`--display-depth` displays the distance to the entry point for each module.

`--display-used-exports` display info about which exports are used in a module.

`--display-max-modules` sets the number for modules displayed in the output (defaults to 15).

`-p` also defines `process.env.NODE_ENV` to `"production"` now.

Loader changes

Changes only relevant for loader authors.

Cacheable

Loaders are now cacheable by default. Loaders must opt-out if they are not cacheable.

```
  // Cacheable loader
  module.exports = function(source) {
-   this.cacheable();
    return source;
  }
```

```
  // Not cacheable loader
  module.exports = function(source) {
+   this.cacheable(false);
    return source;
  }
```

Complex options

webpack 1 only support `JSON.stringify`-able options for loaders. webpack 2 now supports any JS object as loader options.

Using complex options comes with one restriction. You may need to have a `ident` for the option object to make it referenceable by other loaders.

Having an `ident` on the options object means to be able to reference this options object in inline loaders. Here is an example:

```
require("some-loader??by-ident!resource")
```

```
{
  test: /.../,
  loader: "...",
  options: {
    ident: "by-ident",
    magic: () => return Math.random()
  }
}
```

This inline style should not be used by regular code, but it's often used by loader generated code. I. e. the `style-loader` generates a module that `require`s the remaining request (which exports the CSS).

```
// style-loader generated code (simplified)
var addStyle = require("./add-style");
var css = require("-!css-loader?{"modules":true}!postcss-loader??postcss-ident");

addStyle(css);
```

So if you use complex options tell your users about the `ident`.

Code splitting is one of the most compelling features of webpack. It allows you to split your code into various bundles which you can then load on demand — like when a user navigates to a matching route, or on an event from the user. This allows for smaller bundles, and allows you to control resource load prioritization, which if used correctly, can have a major impact on your application load time.

There are mainly two kinds of code splitting that can be accomplished with webpack:

Resource splitting for caching and parallel loads

Vendor code splitting

A typical application can depend on many third party libraries for framework/functionality needs. Unlike application code, code present in these libraries does not change often.

If we keep code from these libraries in its own bundle, separate from the application code, we can leverage the browser's caching mechanism to cache these files for longer durations on the end user's machine.

For this to work, the `hash` portion in the vendor filename must remain constant, regardless of application code changes. Learn how to split vendor/library code using the CommonsChunkPlugin.

CSS splitting

You might also want to split your styles into a separate bundle, independent from application logic. This enhances cacheability of your styles and allows the browser to load the styles in-parallel with your application code, thus preventing a FOUC (flash of unstyled content).

Learn how to split css using the `ExtractTextWebpackPlugin`.

On demand code-splitting

While resource splitting of the previous kind requires the user to specify the split points upfront in the configuration, one can also create dynamic split points in the application code.

This can be used for more granular chunking of code, for example, per our application routes or as per predicted user behaviour. This allows the user to load non-essential assets on demand.

Code splitting with `require.ensure()`

`require.ensure()` is the CommonJS way of including assets asynchronously. By adding `require.ensure([<fileurl>])`, we can define a split point in the code. webpack can then create a separate bundle of all the code inside this split point. Learn how to split code using `require.ensure()`.

?> Document `System.import()`

To bundle CSS files with webpack, import CSS into your JavaScript code like any other module, and use the `css-loader` (which outputs the CSS as JS module), and optionally apply the `ExtractTextWebpackPlugin` (which extracts the bundled CSS and outputs CSS files).

Importing CSS

Import the CSS file like a JavaScript module, for instance in `vendor.js` :

```
import 'bootstrap/dist/css/bootstrap.css';
```

Using `css-loader`

Configure the `css-loader` in `webpack.config.js` as follows:

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

As a result, the CSS is bundled along with your JavaScript.

This has the disadvantage that, you will not be able to utilize the browser's ability to load CSS asynchronously and parallel. Instead, your page will have to wait until your whole JavaScript bundle is loaded, to style itself.

webpack can help with this problem by bundling the CSS separately using the `ExtractTextWebpackPlugin` .

Using `ExtractTextWebpackPlugin`

Install the `ExtractTextWebpackPlugin` plugin as follows

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

To use this plugin, it needs to be added to the `webpack.config.js` file in two steps.

```
module.exports = {
    module: {
         rules: [{
             test: /\.css$/,
-            use: 'css-loader'
+            use: ExtractTextPlugin.extract({
+                use: 'css-loader'
+            })
         }]
     },
+    plugins: [
+        new ExtractTextPlugin('styles.css'),
+    ]
}
```

With above two steps, you can generate a new bundle specifically for all the CSS modules and add them as a separate tag in the `index.html` .

A typical application uses third party libraries for framework/functionality needs. Particular versions of these libraries are used and code here does not change often. However, the application code changes frequently.

Bundling application code with third party code would be inefficient. This is because the browser can cache asset files based on the cache header and files can be cached without needing to call the cdn again if its contents don't change. To take advantage of this, we want to keep the hash of the vendor files constant regardless of application code changes.

We can do this only when we separate the bundles for vendor and application code.

Let's consider a sample application that uses momentjs, a commonly used time formatting library.

Install `moment` as follows in your application directory.

```
npm install --save moment
```

The index file will require `moment` as a dependency and log the current date as follows

index.js

```
var moment = require('moment');
console.log(moment().format());
```

We can bundle the application with webpack using the following config

webpack.config.js

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

module.exports = function(env) {
    return {
        entry: './index.js',
        output: {
            filename: '[chunkhash].[name].js',
            path: path.resolve(__dirname, 'dist')
        }
    }
}
```

On running `webpack` in your application, if you inspect the resulting bundle, you will see that `moment` and `index.js` have been bundled in `bundle.js`.

This is not ideal for the application. If the code in `index.js` changes, then the whole bundle is rebuilt. The browser will have to load a new copy of the new bundle even though most of it hasn't changed at all.

Multiple Entries

Let's try to mitigate this by adding a separate entry point for `moment` and name it `vendor`

webpack.config.js

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

module.exports = function(env) {
    return {
        entry: {
            main: './index.js',
            vendor: 'moment'
        },
        output: {
            filename: '[chunkhash].[name].js',
            path: path.resolve(__dirname, 'dist')
        }
    }
}
```

On running `webpack` now, we see that two bundles have been created. If you inspect these though, you will find that the code for `moment` is present in both the files!

It is for this reason, that we will need to use the CommonsChunkPlugin.

`CommonsChunkPlugin`

This is a pretty complex plugin. It fundamentally allows us to extract all the common modules from different bundles and add them to the common bundle. If a common bundle does not exist, then it creates a new one.

We can modify our webpack config file to use the `CommonsChunkPlugin` as follows

webpack.config.js

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

```
module.exports = function(env) {
    return {
        entry: {
            main: './index.js',
            vendor: 'moment'
        },
        output: {
            filename: '[chunkhash].[name].js',
            path: path.resolve(__dirname, 'dist')
        },
        plugins: [
            new webpack.optimize.CommonsChunkPlugin({
                name: 'vendor' // Specify the common bundle's name.
            })
        ]
    }
}
```

Now run `webpack` on your application. Bundle inspection shows that `moment` code is present only in the vendor bundle.

Implicit Common Vendor Chunk

You can configure a `CommonsChunkPlugin` instance to only accept vendor libraries.

webpack.config.js

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

module.exports = function() {
    return {
        entry: {
            main: './index.js'
        },
        output: {
            filename: '[chunkhash].[name].js',
            path: path.resolve(__dirname, 'dist')
        },
        plugins: [
            new webpack.optimize.CommonsChunkPlugin({
                name: 'vendor',
                minChunks: function (module) {
                   // this assumes your vendor imports exist in the node_modules d
irectory
                   return module.context && module.context.indexOf('node_modules')
```

```
  !== -1;
                }
            })
        ]
    };
}
```

Manifest File

But, if we change application code and run `webpack` again, we see that the hash for the vendor file changes. Even though we achieved separate bundles for `vendor` and `main` bundles, we see that the `vendor` bundle changes when the application code changes. This means that we still don't reap the benefits of browser caching because the hash for vendor file changes on every build and the browser will have to reload the file.

The issue here is that on every build, webpack generates some webpack runtime code, which helps webpack do it's job. When there is a single bundle, the runtime code resides in it. But when multiple bundles are generated, the runtime code is extracted into the common module, here the `vendor` file.

To prevent this, we need extract out the runtime into a separate manifest file. Even though we are creating another bundle, the overhead is offset by the long term caching benefits that we obtain on the `vendor` file.

webpack.config.js

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

module.exports = function(env) {
    return {
        entry: {
            main: './index.js',
            vendor: 'moment'
        },
        output: {
            filename: '[chunkhash].[name].js',
            path: path.resolve(__dirname, 'dist')
        },
        plugins: [
            new webpack.optimize.CommonsChunkPlugin({
                names: ['vendor', 'manifest'] // Specify the common bundle's name.
            })
        ]
    }
```

```
};
```

With the above webpack config, we see three bundles being generated. `vendor`, `main` and `manifest` bundles.

T> Note that long-term bundle caching is achieved with content-based hashing policy `chunkhash`. Learn more about caching.

In this section, we will discuss how webpack splits code using `require.ensure()`.

`require.ensure()`

webpack statically parses for `require.ensure()` in the code while building and adds the modules here into a separate chunk. This new chunk is loaded on demand by webpack through jsonp.

The syntax is as follows

```
require.ensure(dependencies: String[], callback: function(require), chunkName: Str
ing)
```

dependencies

This is an array of strings where we can declare all the modules that need to be made available before all the code in the callback function can be executed.

callback

This is the callback function that webpack will execute once the dependencies are loaded. An implementation of the require object is sent as a parameter to this function. This is so that, we can further `require()` the dependencies and any other modules for execution.

chunkName

The chunkName is the name given to the chunk created by this particular `require.ensure()`. By giving the same name at different split points of `require.ensure()`, we can make sure all the dependencies are collectively put in the same bundle.

Let us consider the following project

```
\\ file structure
    |
    js --|
    |    |-- entry.js
    |    |-- a.js
    |    |-- b.js
    webpack.config.js
```

```
    |
    dist
```

```
\\ entry.js

require('a');
require.ensure([], function(require){
    require('b');
});

\\ a.js
console.log('***** I AM a *****');

\\ b.js
console.log('***** I AM b *****');
```

```
\\ webpack.config.js
var path = require('path');

module.exports = function(env) {
    return {
        entry: './js/entry.js',
        output: {
            filename: 'bundle.js',
            path: path.resolve(__dirname, 'dist')
        }
    }
}
```

On running webpack on this project, we find that webpack has created two new bundles, `bundle.js` and `0.bundle.js`.

`entry.js` and `a.js` are bundled in `bundle.js`.

`b.js` is bundled in `0.bundle.js`.

W> `require.ensure` relies on `Promises` internally. If you use `require.ensure` with older browsers, remember to shim `Promise.` es6-promise polyfill.

Gotchas for `require.ensure()`

Empty Array as Parameter

```
require.ensure([], function(require){
    require('./a.js');
});
```

The above code ensures that a split point is created and `a.js` is bundled separately by webpack.

Dependencies as Parameter

```
require.ensure(['./a.js'], function(require) {
    require('./b.js');
});
```

In the above code, `a.js` and `b.js` are bundled together and split from the main bundle. But only the contents of `b.js` are executed. The contents of `a.js` are only made available and not executed. To execute `a.js`, we will have to require it in a sync manner like `require('./a.js')` for the JavaScript to get executed.

This page explains how to generate production builds with webpack.

The automatic way

Running `webpack -p` (or equivalently `webpack --optimize-minimize --define process.env.NODE_ENV="'production'"`). This performs the following steps:

- Minification using `UglifyJsPlugin`
- Runs the `LoaderOptionsPlugin`, see its documentation
- Sets the Node environment variable

Minification

webpack comes with `UglifyJsPlugin`, which runs UglifyJS in order to minimize the output. The plugin supports all of UglifyJS options. Specifying `--optimize-minimize` on the command line, the following plugin configuration is added:

```
// webpack.config.js
const webpack = require('webpack');

module.exports = {
  /*...*/
  plugins:[
    new webpack.optimize.UglifyJsPlugin({
      sourceMap: options.devtool && (options.devtool.indexOf("sourcemap") >= 0 ||
options.devtool.indexOf("source-map") >= 0)
    })
  ]
};
```

Thus, depending on the devtool options, Source Maps are generated.

Source Maps

We encourage you to have Source Maps enabled in production. They are useful for debugging and to run benchmark tests. webpack can generate inline Source Maps included in the bundles or separated files.

In your configuration, use the `devtool` object to set the Source Map type. We currently support seven types of Source Maps. You can find more information about them in our configuration documentation page.

One of the good options to go is using `cheap-module-source-map` which simplifies the Source Maps to a single mapping per line.

Node environment variable

Running `webpack -p` (or `--define process.env.NODE_ENV="production"`) invokes the `DefinePlugin` in the following way:

```
// webpack.config.js
const webpack = require('webpack');

module.exports = {
  /*...*/
  plugins:[
    new webpack.DefinePlugin({
      'process.env.NODE_ENV': JSON.stringify('production')
    })
  ]
};
```

The `DefinePlugin` performs search-and-replace operations on the original source code. Any occurrence of `process.env.NODE_ENV` in the imported code is replaced by `"production"`. Thus, checks like `if (process.env.NODE_ENV !== 'production') console.log('...')` are evaluated to `if (false) console.log('...')` and finally minified away using `UglifyJS`.

T> Technically, `NODE_ENV` is a system environment variable that Node.js exposes into running scripts. It is used by convention to determine development-vs-production behavior, by both server tools, build scripts, and client-side libraries. Contrary to expectations, `process.env.NODE_ENV` is not set to `"production"` **within** the build script `webpack.config.js`, see #2537. Thus, conditionals like `process.env.NODE_ENV === 'production' ? '[name].[hash].bundle.js' : '[name].bundle.js'` do not work as expected. See how to specify the environment in webpack configuration.

The manual way: Configuring webpack for multiple environments

When we do have multiple configurations in mind for different environments, the easiest way is to write separate js files for each environment. For example:

dev.js

```
module.exports = function (env) {
  return {
    devtool: 'cheap-module-source-map',
    output: {
        path: path.join(__dirname, '/../dist/assets'),
        filename: '[name].bundle.js',
        publicPath: publicPath,
        sourceMapFilename: '[name].map'
    },
    devServer: {
        port: 7777,
        host: 'localhost',
        historyApiFallback: true,
        noInfo: false,
        stats: 'minimal',
        publicPath: publicPath
    }
  }
}
```

prod.js

```
module.exports = function (env) {
  return {
    output: {
        path: path.join(__dirname, '/../dist/assets'),
        filename: '[name].bundle.js',
        publicPath: publicPath,
        sourceMapFilename: '[name].map'
    },
    plugins: [
        new webpack.LoaderOptionsPlugin({
            minimize: true,
            debug: false
        }),
        new webpack.optimize.UglifyJsPlugin({
            beautify: false,
            mangle: {
                screw_ie8: true,
                keep_fnames: true
            },
            compress: {
                screw_ie8: true
            },
            comments: false
        })
    ]
  }
}
```

Have the following snippet in your webpack.config.js:

```
function buildConfig(env) {
  return require('./config/' + env + '.js')({ env: env })
}

module.exports = buildConfig;
```

And from our package.json, where we build our application using webpack, the command goes like this:

```
 "build:dev": "webpack --env=dev --progress --profile --colors",
 "build:dist": "webpack --env=prod --progress --profile --colors",
```

You could see that we passed the environment variable to our webpack.config.js file. From there we used a simple switch-case to build for the environment we passed by simply loading the right js file.

An advanced approach would be to have a base configuration file, put in all common functionalities, and then have environment specific files and simply use 'webpack-merge' to merge them. This would help to avoid code repetitions. For example, you could have all your base configurations like resolving your js, ts, png, jpeg, json and so on.. in a common base file as follows:

base.js

```
module.exports = function() {
    return {
        entry: {
            'polyfills': './src/polyfills.ts',
            'vendor': './src/vendor.ts',
            'main': './src/main.ts'

        },
        output: {
            path: path.join(__dirname, '/../dist/assets'),
            filename: '[name].bundle.js',
            publicPath: publicPath,
            sourceMapFilename: '[name].map'
        },
        resolve: {
            extensions: ['', '.ts', '.js', '.json'],
            modules: [path.join(__dirname, 'src'), 'node_modules']

        },
```

```
        module: {
            loaders: [{
                test: /\.ts$/,
                loaders: [
                    'awesome-typescript-loader',
                    'angular2-template-loader'
                ],
                exclude: [/\.(spec|e2e)\.ts$/]
            }, {
                test: /\.css$/,
                loaders: ['to-string-loader', 'css-loader']
            }, {
                test: /\.(jpg|png|gif)$/,
                loader: 'file-loader'
            }, {
                test: /\.(woff|woff2|eot|ttf|svg)$/,
                loader: 'url-loader?limit=100000'
            }],
        },
        plugins: [
            new ForkCheckerPlugin(),

            new webpack.optimize.CommonsChunkPlugin({
                name: ['polyfills', 'vendor'].reverse()
            }),
            new HtmlWebpackPlugin({
                template: 'src/index.html',
                chunksSortMode: 'dependency'
            })
        ],
    };
}
```

And then merge this base config with an environment specific configuration file using 'webpack-merge'. Let us look at an example where we merge our prod file, mentioned above, with this base config file using 'webpack-merge':

prod.js (updated)

```
const webpackMerge = require('webpack-merge');

const commonConfig = require('./base.js');

module.exports = function(env) {
    return webpackMerge(commonConfig(), {
        plugins: [
            new webpack.LoaderOptionsPlugin({
                minimize: true,
                debug: false
```

```
            }),
            new webpack.DefinePlugin({
                'process.env': {
                    'NODE_ENV': JSON.stringify('prod')
                }
            }),
            new webpack.optimize.UglifyJsPlugin({
                beautify: false,
                mangle: {
                    screw_ie8: true,
                    keep_fnames: true
                },
                compress: {
                    screw_ie8: true
                },
                comments: false
            })
        ]
    })
}
```

You will notice three major updates to our 'prod.js' file.

- 'webpack-merge' with the 'base.js'.
- We have move 'output' property to 'base.js'. Just to stress on that point that our output property, here, is common across all our environments and that we refactored our 'prod.js' and moved it to our 'base.js', the common configuration file.
- We have defined the 'process.env.NODE_ENV' to be 'prod' using the 'DefinePlugin'. Now across the application 'process.env.NODE_ENV' would have the value, 'prod', when we build our application for production environment. Likewise we can manage various variables of our choice, specific to environments this way.

The choice of what is going to be common across all your environments is up to you, however. We have just demonstrated a few that could typically be common across environments when we build our application.

To enable long-term caching of static resources produced by webpack:

1. Use `[chunkhash]` to add a content-dependent cache-buster to each file.
2. Extract the webpack manifest into a separate file.
3. Ensure that the entry point chunk containing the bootstrapping code doesn't change hash over time for the same set of dependencies.

   For even more optimized setup:

4. Use compiler stats to get the file names when requiring resources in HTML.
5. Generate the chunk manifest JSON and inline it into the HTML page before loading resources.

The problem

Each time something needs to be updated in our code, it has to be re-deployed on the server and then re-downloaded by all clients. This is clearly inefficient since fetching resources over the network can be slow. This is why browsers cache static resources.

The way it works has a pitfall: If we don't change filenames of our resources when deploying a new version, the browser might think it hasn't been updated and client will get a cached version of it.

A simple way to tell the browser to download a newer version is to alter the asset's file name. In a pre-webpack era we used to add a build number to the filenames as a parameter and then increment it:

```
application.js?build=1
application.css?build=1
```

It is even easier to do with webpack. Each webpack build generates a unique hash which can be used to compose a filename, by including output placeholders. The following example config will generate 2 files (1 per entry) with hashes in filenames:

```
// webpack.config.js
const path = require("path");

module.exports = {
  entry: {
    vendor: "./src/vendor.js",
    main: "./src/index.js"
  },
```

```
  output: {
    path: path.join(__dirname, "build"),
    filename: "[name].[hash].js"
  }
};
```

Running webpack with this config will produce the following output:

```
Hash: 2a6c1fee4b5b0d2c9285
Version: webpack 2.2.0
Time: 62ms
                         Asset     Size  Chunks             Chunk Names
vendor.2a6c1fee4b5b0d2c9285.js  2.58 kB       0  [emitted]  vendor
  main.2a6c1fee4b5b0d2c9285.js  2.57 kB       1  [emitted]  main
   [0] ./src/index.js 63 bytes {1} [built]
   [1] ./src/vendor.js 63 bytes {0} [built]
```

But the problem here is, builds after *any file update* will update all filenames and clients will have to re-download all application code. So how can we guarantee that clients always get the latest versions of assets without re-downloading all of them?

Generating unique hashes for each file

What if we could produce the same filename, if the contents of the file did not change between builds? For example, it would be unnecessary to re-download a vendor file, when no dependencies have been updated, only application code.

webpack allows you to generate hashes depending on file contents, by replacing the placeholder `[hash]` with `[chunkhash]`. Here is the updated config:

```
module.exports = {
  /*...*/
  output: {
    /*...*/
-   filename: "[name].[hash].js"
+   filename: "[name].[chunkhash].js"
  }
};
```

This config will also create 2 files, but in this case, each file will get its own unique hash.

```
Hash: cfba4af36e2b11ef15db
Version: webpack 2.2.0
```

```
Time: 66ms
                         Asset     Size  Chunks             Chunk Names
vendor.50cfb8f89ce2262e5325.js  2.58 kB       0  [emitted]  vendor
  main.70b594fe8b07bcedaa98.js  2.57 kB       1  [emitted]  main
   [0] ./src/index.js 63 bytes {1} [built]
   [1] ./src/vendor.js 63 bytes {0} [built]
```

T> Don't use [chunkhash] in development since this will increase compilation time. Separate development and production configs and use [name].js for development and [name].[chunkhash].js in production.

Get filenames from webpack compilation stats

When working in development mode, you just reference JavaScript files by entry point name in your HTML.

```
<script src="vendor.js"></script>
<script src="main.js"></script>
```

Although, each time we build for production, we'll get different file names. Something, that looks like this:

```
<script src="vendor.50cfb8f89ce2262e5325.js"></script>
<script src="main.70b594fe8b07bcedaa98.js"></script>
```

In order to reference a correct file in the HTML, we'll need information about our build. This can be extracted from webpack compilation stats by using this plugin:

```
// webpack.config.js
const path = require("path");

module.exports = {
  /*...*/
  plugins: [
    function() {
      this.plugin("done", function(stats) {
        require("fs").writeFileSync(
          path.join(__dirname, "build", "stats.json"),
          JSON.stringify(stats.toJson()));
      });
    }
```

```
  ]
};
```

Alternatively, just use one of these plugins to export JSON files:

- https://www.npmjs.com/package/webpack-manifest-plugin
- https://www.npmjs.com/package/assets-webpack-plugin

A sample output when using `WebpackManifestPlugin` in our config looks like:

```
{
  "main.js": "main.155567618f4367cd1cb8.js",
  "vendor.js": "vendor.c2330c22cd2decb5da5a.js"
}
```

Deterministic hashes

To minimize the size of generated files, webpack uses identifiers instead of module names. During compilation, identifiers are generated, mapped to chunk filenames and then put into a JavaScript object called *chunk manifest*. To generate identifiers that are preserved over builds, webpack supply the `NamedModulesPlugin` (recommended for development) and `HashedModuleIdsPlugin` (recommended for production).

?> When exist, link to `NamedModulesPlugin` and `HashedModuleIdsPlugin` docs pages

?> Describe how the option `recordsPath` option works

The chunk manifest (along with bootstrapping/runtime code) is then placed into the entry chunk and it is crucial for webpack-packaged code to work.

T> Separate your vendor and application code with CommonsChunkPlugin and create an explicit vendor chunk to prevent it from changing too often. When `CommonsChunkPlugin` is used, the runtime code is moved to the *last* common entry.

The problem with this, is the same as before: Whenever we change any part of the code it will, even if the rest of its contents wasn't altered, update our entry chunk to include the new manifest. This in turn, will lead to a new hash and dismiss the long-term caching.

To fix that, we should use `ChunkManifestWebpackPlugin`, which will extract the manifest to a separate JSON file. This replaces the chunk manifest with a variable in the webpack runtime. But we can do even better; we can extract the runtime into a separate entry by

using `CommonsChunkPlugin`. Here is an updated `webpack.config.js` which will produce the manifest and runtime files in our build directory:

```
// webpack.config.js
var ChunkManifestPlugin = require("chunk-manifest-webpack-plugin");

module.exports = {
  /*...*/
  plugins: [
    /*...*/
    new webpack.optimize.CommonsChunkPlugin({
      name: ["vendor", "manifest"], // vendor libs + extracted manifest
      minChunks: Infinity,
    }),
    /*...*/
    new ChunkManifestPlugin({
      filename: "chunk-manifest.json",
      manifestVariable: "webpackManifest"
    })
  ]
};
```

As we removed the manifest from the entry chunk, now it's our responsibility to provide webpack with it. The `manifestVariable` option in the example above is the name of the global variable where webpack will look for the manifest JSON. This *should be defined before we require our bundle in HTML*. This is achieved by inlining the contents of the JSON in HTML. Our HTML head section should look like this:

```
<html>
  <head>
    <script>
    //<![CDATA[
    window.webpackManifest = {"0":"main.5f020f80c23aa50ebedf.js","1":"vendor.81adc
64d405c8b218485.js"}
    //]]>
    </script>
  </head>
  <body>
  </body>
</html>
```

At the end of the day, the hashes for the files should be based on the file content. For this use webpack-chunk-hash or webpack-md5-hash.

So the final `webpack.config.js` should look like this:

```
var path = require("path");
var webpack = require("webpack");
var ChunkManifestPlugin = require("chunk-manifest-webpack-plugin");
var WebpackChunkHash = require("webpack-chunk-hash");

module.exports = {
  entry: {
    vendor: "./src/vendor.js", // vendor reference file(s)
    main: "./src/index.js" // application code
  },
  output: {
    path: path.join(__dirname, "build"),
    filename: "[name].[chunkhash].js",
    chunkFilename: "[name].[chunkhash].js"
  },
  plugins: [
    new webpack.optimize.CommonsChunkPlugin({
      name: ["vendor", "manifest"], // vendor libs + extracted manifest
      minChunks: Infinity,
    }),
    new webpack.HashedModuleIdsPlugin(),
    new WebpackChunkHash(),
    new ChunkManifestPlugin({
      filename: "chunk-manifest.json",
      manifestVariable: "webpackManifest"
    })
  ]
};
```

Using this config the vendor chunk should not be changing its hash, unless you update its code or dependencies. Here is a sample output for 2 runs with `moduleB.js` being changed between the runs:

```
> node_modules/.bin/webpack

Hash: f0ae5bf7c6a1fd3b2127
Version: webpack 2.2.0
Time: 102ms
                          Asset       Size  Chunks             Chunk Names
    main.9ebe4bf7d99ffc17e75f.js  509 bytes    0, 2  [emitted]  main
  vendor.81adc64d405c8b218485.js  159 bytes    1, 2  [emitted]  vendor
             chunk-manifest.json   73 bytes          [emitted]
manifest.d41d8cd98f00b204e980.js    5.56 kB       2  [emitted]  manifest
```

```
> node_modules/.bin/webpack

Hash: b5fb8e138b039ab515f3
```

```
Version: webpack 2.2.0
Time: 87ms
                           Asset       Size  Chunks             Chunk Names
    main.5f020f80c23aa50ebedf.js  521 bytes    0, 2  [emitted]  main
  vendor.81adc64d405c8b218485.js  159 bytes    1, 2  [emitted]  vendor
             chunk-manifest.json   73 bytes          [emitted]
manifest.d41d8cd98f00b204e980.js    5.56 kB       2  [emitted]  manifest
```

Notice that **vendor chunk has the same filename**, and **so does the manifest** since we've extracted the manifest chunk!

Manifest inlining

Inlining the chunk manifest and webpack runtime (to prevent extra HTTP requests), depends on your server setup. There is a nice walkthrough for Rails-based projects. For server-side rendering in Node.js you can use webpack-isomorphic-tools.

T> If your application doesn't rely on any server-side rendering, it's often enough to generate a single `index.html` file for your application. To do so, use i.e. `HtmlWebpackPlugin` in combination with `ScriptExtHtmlWebpackPlugin` or `InlineManifestWebpackPlugin`. It will simplify the setup dramatically.

References

- https://medium.com/@okonetchnikov/long-term-caching-of-static-assets-with-webpack-1ecb139adb95#.vtwnssps4
- https://gist.github.com/sokra/ff1b0290282bfa2c037bdb6dcca1a7aa
- https://github.com/webpack/webpack/issues/1315
- https://github.com/webpack/webpack.js.org/issues/652
- https://presentations.survivejs.com/advanced-webpack/

On this page we'll explain how to get started with developing and how to choose one of three tools to develop. It is assumed you already have a webpack configuration file.

W> Never use any of these tools in production. Ever.

Adjusting Your Text Editor

Some text editors have a "safe write" feature and enable this by default. As a result, saving a file will not always result in a recompile.

Each editor has a different way of disabling this. For the most common ones:

- **Sublime Text 3** - Add `"atomic_save": false` to your user preferences.
- **IntelliJ** - use search in the preferences to find "safe write" and disable it.
- **Vim** - add `:set backupcopy=yes` in your settings.
- **WebStorm** - uncheck `Use "safe write"` in Preferences > Appearance & Behavior > System Settings

Source Maps

When a JavaScript exception occurs, you'll often want to know what file and line is generating this error. Since webpack outputs files into one or more bundles, it can be inconvenient to trace the file.

Source maps intend to fix this problem. There are a lot of different options - each with their own advantages and disadvantages. To get started, we'll use this one:

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

Choosing a Tool

webpack can be used with **watch mode**. In this mode webpack will watch your files, and recompile when they change. **webpack-dev-server** provides an easy to use development server with fast live reloading. If you already have a development server and want full flexibility, **webpack-dev-middleware** can be used as middleware.

webpack-dev-server and webpack-dev-middleware use in-memory compilation, meaning that the bundle will not be saved to disk. This makes compiling faster and results in less mess on your file system.

In most cases **you'll want to use webpack-dev-server**, since it's the easiest to get started with and offers much functionality out-of-the-box.

webpack Watch Mode

webpack's watch mode watches files for changes. If any change is detected, it'll run the compilation again.

We also want a nice progress bar while it's compiling. Let's run the command:

```
webpack --progress --watch
```

Make a change in one of your files and hit save. You should see that it's recompiling.

Watch mode makes no assumptions about a server, so you will need to provide your own. An easy server is `serve`. After installing (`npm i serve -g`), you can execute it in the directory where the outputted files are:

```
serve
```

After each compilation, you will need to manually refresh your browser to see the changes.

Watch Mode with Chrome DevTools Workspaces

If you set up Chrome to persist changes when saving from the *Sources* panel so you don't have to refresh the page, you will have to setup webpack to use

```
devtool: "inline-source-map"
```

to continue editing and saving your changes from Chrome or source files.

There are some *gotchas* about using workspaces with watch:

- Large chunks (such as a common chunk that is over 1MB) that are rebuilt could cause the page to blank, which will force you to refresh the browser.

- Smaller chunks will be faster to build than larger chunks since `inline-source-map` is slower due to having to base64 encode the original source code.

webpack-dev-server

webpack-dev-server provides you with a server and live reloading. This is easy to setup.

To prepare, make sure you have a `index.html` file that points to your bundle. Assuming that `output.filename` is `bundle.js` :

```
<script src="/bundle.js"></script>
```

Start with installing `webpack-dev-server` from npm:

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

When it's done installing, you should be able to use `webpack-dev-server` like this:

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

T> If your console says it can't find the command, try running `node_modules/.bin/webpack-dev-server` . Optimally you would add the command to your `package.json` , like this: `"scripts": { "start": "webpack-dev-server" }` .

The command above should automatically open your browser on `http://localhost:8080` .

Make a change in one of your files and hit save. You should see that the console is recompiling. After that's done, the page should be refreshed. If nothing happens in the console, you may need to fiddle with `watchOptions` .

Now you have live reloading working, you can take it even a step further: Hot Module Replacement. This is an interface that makes it possible to swap modules **without a page refresh**. Find out how to configure HMR.

By default **inline mode** is used. This mode injects the client - needed for live reloading and showing build errors - in your bundle. With inline mode you will get build errors and warnings in your DevTools console.

webpack-dev-server can do many more things such as proxying requests to your backend server. For more configuration options, see the **devServer documentation**.

webpack-dev-middleware

webpack-dev-middleware works for connect-based middleware stacks. This can be useful if you already have a Node.js server or if you want to have full control over the server.

The middleware will cause webpack to compile files in-memory. When a compilation is running, it will delay the request to a file until the compilation is done.

W> This is intended for advanced users. webpack-dev-server is much easier to use.

Start with installing the dependencies from npm:

```
npm install express webpack-dev-middleware --save-dev
```

After installing, you can use the middleware like this:

```
var express = require("express");
var webpackDevMiddleware = require("webpack-dev-middleware");
var webpack = require("webpack");
var webpackConfig = require("./webpack.config");

var app = express();
var compiler = webpack(webpackConfig);

app.use(webpackDevMiddleware(compiler, {
  publicPath: "/" // Same as `output.publicPath` in most cases.
}));

app.listen(3000, function () {
  console.log("Listening on port 3000!");
});
```

Depending on what you've used in `output.publicPath` and `output.filename`, your bundle should now be available on `http://localhost:3000/bundle.js`.

By default, **watch mode** is used. It's also possible to use **lazy mode**, which will only recompile on a request to the entry point.

To compile only on a request to the entry `bundle.js`:

```
app.use(webpackDevMiddleware(compiler, {
  lazy: true,
  filename: "bundle.js" // Same as `output.filename` in most cases.
}));
```

There are many more options you can use. For all configuration options, see the **devServer documentation**.

References

- SurviveJS - Automatic Browser Refresh
- Webpack your Chrome DevTools Workspaces

If you have a more advanced project and use Vagrant to run your development environment in a Virtual Machine, you'll often want to also run webpack in the VM.

Configuring the Project

To start, make sure that the `Vagrantfile` has a static IP;

```
Vagrant.configure("2") do |config|
  config.vm.network :private_network, ip: "10.10.10.61"
end
```

Next, install webpack and webpack-dev-server in your project;

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

Make sure to have a `webpack.config.js` file. If you haven't already, use this as a minimal example to get started:

```
module.exports = {
  context: __dirname,
  entry: "./app.js"
};
```

And create a `index.html` file. The script tag should point to your bundle. If `output.filename` is not specified in the config, this will be `bundle.js`.

```
<!DOCTYPE html>
<html>
  <head>
    <script src="/bundle.js" charset="utf-8"></script>
  </head>
  <body>
    <h2>Heey!</h2>
  </body>
</html>
```

Note that you also need to create an `app.js` file.

Running the Server

Now, let's run the server:

```
webpack-dev-server --host 0.0.0.0 --public 10.10.10.61:8080 --watch-poll
```

By default the server will only be accessible from localhost. We'll be accessing it from our host PC, so we need to change `--host` to allow this.

webpack-dev-server will include a script in your bundle that connects to a WebSocket to reload when a change in any of your files occurs. The `--public` flag makes sure the script knows where to look for the WebSocket. The server will use port `8080` by default, so we should also specify that here.

`--watch-poll` makes sure that webpack can detect changes in your files. By default webpack listens to events triggered by the filesystem, but VirtualBox has many problems with this.

The server should be accessible on `http://10.10.10.61:8080` now. If you make a change in `app.js`, it should live reload.

Advanced Usage with nginx

To mimic a more production-like environment, it is also possible to proxy the webpack-dev-server with nginx.

In your nginx config file, add the following:

```
server {
  location / {
    proxy_pass http://127.0.0.1:8080;
    proxy_http_version 1.1;
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection "upgrade";
    error_page 502 @start-webpack-dev-server;
  }

  location @start-webpack-dev-server {
    default_type text/plain;
    return 502 "Please start the webpack-dev-server first.";
  }
}
```

The `proxy_set_header` lines are important, because they allow the WebSockets to work correctly.

The command to start webpack-dev-server can then be changed to this:

```
webpack-dev-server --public 10.10.10.61 --watch-poll
```

This makes the server only accessible on `127.0.0.1`, which is fine, because nginx takes care of making it available on your host PC.

Conclusion

We made the Vagrant box accessible from a static IP, and then made webpack-dev-server publicly accessible so it is reachable from a browser. We then tackled a common problem that VirtualBox doesn't send out filesystem events, causing the server to not reload on file changes.

> es6 modules
>
> commonjs
>
> amd

require with expression

A context is created if your request contains expressions, so the **exact** module is not known on compile time.

Example:

```
require("./template/" + name + ".ejs");
```

webpack parses the `require()` call and extracts some information:

```
Directory: ./template
Regular expression: /^.*\.ejs$/
```

context module

A context module is generated. It contains references to **all modules in that directory** that can be required with a request matching the regular expression. The context module contains a map which translates requests to module ids.

Example:

```
{
    "./table.ejs": 42,
    "./table-row.ejs": 43,
    "./directory/folder.ejs": 44
}
```

The context module also contains some runtime logic to access the map.

This means dynamic requires are supported but will cause all possible modules to be included in the bundle.

`require.context`

You can create your own context with the `require.context()` function. It allows you to pass in a directory to search, a flag indicating whether subdirectories should be searched too, and a regular expression to match files against.

webpack parses for `require.context()` in the code while building.

The syntax is as follows:

```
require.context(directory, useSubdirectories = false, regExp = /^\.\//)
```

Examples:

```
require.context("./test", false, /\.test\.js$/);
// a context with files from the test directory that can be required with a reques
t endings with `.test.js`.
```

```
require.context("../", true, /\.stories\.js$/);
// a context with all files in the parent folder and descending folders ending wit
h `.stories.js`.
```

context module API

A context module exports a (require) function that takes one argument: the request.

The exported function has 3 properties: `resolve`, `keys`, `id`.

- `resolve` is a function and returns the module id of the parsed request.
- `keys` is a function that returns an array of all possible requests that the context module can handle.

  This can be useful if you want to require all files in a directory or matching a pattern, Example:

  ```
  function importAll (r) {
    r.keys().forEach(r);
  }
  importAll(require.context('../components/', true, /\.js$/));
  ```

  ```
  var cache = {};
  function importAll (r) {
    r.keys().forEach(key => cache[key] = r(key));
  }
  ```

```
importAll(require.context('../components/', true, /\.js$/));
// At build-time cache will be populated with all required modules.
```

- `id` is the module id of the context module. This may be useful for `module.hot.accept`.

`webpack` as a module bundler can understand modules written as ES2015 modules, CommonJS or AMD. But many times, while using third party libraries, we see that they expect dependencies which are global, AKA `$` for `jquery`. They might also be creating global variables which need to be exported. Here we will see different ways to help webpack understand these **broken modules**.

Prefer unminified CommonJS/AMD files over bundled `dist` versions.

Most modules link the `dist` version in the `main` field of their `package.json`. While this is useful for most developers, for webpack it is better to alias the src version because this way webpack is able to optimize dependencies better. However in most cases `dist` works fine as well.

```
// webpack.config.js

module.exports = {
    ...
    resolve: {
        alias: {
            jquery: "jquery/src/jquery"
        }
    }
};
```

`ProvidePlugin`

The `ProvidePlugin` makes a module available as a variable in every other module required by `webpack`. The module is required only if you use the variable. Most legacy modules rely on the presence of specific globals, like jQuery plugins do on `$` or `jQuery`. In this scenario, you can configure webpack to prepend `var $ = require("jquery")` every time it encounters the global `$` identifier.

```
module.exports = {
  plugins: [
    new webpack.ProvidePlugin({
      $: 'jquery',
      jQuery: 'jquery'
    })
  ]
};
```

`imports-loader`

`imports-loader` inserts necessary globals into the required legacy module. For example, Some legacy modules rely on `this` being the `window` object. This becomes a problem when the module is executed in a CommonJS context where `this` equals `module.exports`. In this case you can override `this` using the `imports-loader`.

webpack.config.js

```
module.exports = {
  module: {
    rules: [{
      test: require.resolve("some-module"),
      use: 'imports-loader?this=>window'
    }]
  }
};
```

There are modules that support different module styles, like AMD, CommonJS and legacy. However, most of the time they first check for `define` and then use some quirky code to export properties. In these cases, it could help to force the CommonJS path by setting `define = false`:

webpack.config.js

```
module.exports = {
  module: {
    rules: [{
      test: require.resolve("some-module"),
      use: 'imports-loader?define=>false'
    }]
  }
};
```

`exports-loader`

Let's say a library creates a global variable that it expects its consumers to use; In this case, we can use `exports-loader`, to export that global variable in CommonJS format. For instance, in order to export `file` as `file` and `helpers.parse` as `parse`:

webpack.config.js

```
module.exports = {
  module: {
    rules: [{
      test: require.resolve("some-module"),
      use: 'exports-loader?file,parse=helpers.parse'
      // adds below code the file's source:
      //  exports["file"] = file;
      //  exports["parse"] = helpers.parse;
    }]
  }
};
```

`script-loader`

The `script-loader` evaluates code in the global context, just like you would add the code into a `script` tag. In this mode, every normal library should work. `require`, `module`, etc. are undefined.

W> The file is added as string to the bundle. It is not minimized by `webpack`, so use a minimized version. There is also no dev tool support for libraries added by this loader.

Assuming you have a `legacy.js` file containing …

```
GLOBAL_CONFIG = {};
```

… using the `script-loader` …

```
require('script-loader!legacy.js');
```

… basically yields:

```
eval("GLOBAL_CONFIG = {};");
```

`noParse` option

When there is no AMD/CommonJS version of the module and you want to include the `dist`, you can flag this module as `noParse`. Then `webpack` will just include the module without parsing it, which can be used to improve the build time.

W> Any feature requiring the AST, like the `ProvidePlugin`, will not work.

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

webpack is a tool which can be used to bundle application code and also to bundle library code. If you are the author of a JavaScript library and are looking to streamline your bundle strategy then this document will help you.

Author a Library

Let's assume that you are writing a small library `webpack-numbers` allowing to convert numbers 1 to 5 from their numeric to a textual representation and vice-versa. The implementation makes use of ES2015 modules, and might look like this:

src/index.js

```
import _ from 'lodash';
import numRef from './ref.json';

export function numToWord(num) {
    return _.reduce(numRef, (accum, ref) => {
        return ref.num === num ? ref.word : accum;
    }, '');
};

export function wordToNum(word) {
    return _.reduce(numRef, (accum, ref) => {
        return ref.word === word && word.toLowerCase() ? ref.num : accum;
    }, -1);
};
```

The usage spec for the library will be as follows.

```
import * as webpackNumbers from 'webpack-numbers';

...
webpackNumbers.wordToNum('Two') // output is 2
...

// CommonJS modules

var webpackNumbers = require('webpack-numbers');

...
webpackNumbers.numToWord(3); // output is Three
...
```

```
// Or as a script tag
```

```
<html>
...
<script src="https://unpkg.com/webpack-numbers"></script>
<script>
    ...
    /* webpackNumbers is available as a global variable */
    webpackNumbers.wordToNum('Five') //output is 5
    ...
</script>
</html>
```

For full library configuration and code please refer to webpack-library-example

Configure webpack

Now the agenda is to bundle this library

- Without bundling `lodash` but requiring it to be loaded by the consumer.
- Name of the library is `webpack-numbers` and the variable is `webpackNumbers`.
- Library can be imported as `import webpackNumbers from 'webpack-numbers'` or `require('webpack-numbers')`.
- Library can be accessed through global variable `webpackNumbers` when included through `script` tag.
- Library can be accessed inside Node.js.

Add webpack

Add basic webpack configuration.

webpack.config.js

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

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

This adds basic configuration to bundle the library.

Add `externals`

Now, if you run `webpack`, you will find that a largish bundle file is created. If you inspect the file, you will find that lodash has been bundled along with your code. It would be unnecessary for your library to bundle a library like `lodash`. Hence you would want to give up control of this external library to the consumer of your library.

This can be done using the `externals` configuration as

webpack.config.js

```
module.exports = {
    ...
    externals: {
        "lodash": {
            commonjs: "lodash",
            commonjs2: "lodash",
            amd: "lodash",
            root: "_"
        }
    }
    ...
};
```

This means that your library expects a dependency named `lodash` to be available in the consumer's environment.

Add `libraryTarget`

For widespread use of the library, we would like it to be compatible in different environments, i. e. CommonJS, AMD, Node.js and as a global variable.

To make your library available for reuse, add `library` property in webpack configuration.

webpack.config.js

```
module.exports = {
    ...
    output: {
        ...
        library: 'webpackNumbers'
    }
    ...
};
```

This makes your library bundle to be available as a global variable when imported. To make the library compatible with other environments, add `libraryTarget` property to the config.

webpack.config.js

```
module.exports = {
    ...
    output: {
        ...
        library: 'webpackNumbers',
        libraryTarget: 'umd' // Possible value - amd, commonjs, commonjs2, commonjs-module, this, var
    }
    ...
};
```

If `library` is set and `libraryTarget` is not, `libraryTarget` defaults to `var` as specified in the config reference.

Final Steps

Tweak your production build using webpack.

Add the path to your generated bundle as the package's main file in `package.json`

package.json

```
{
    ...
    "main": "dist/webpack-numbers.js",
    "module": "src/index.js", // To add as standard module as per https://github.com/dherman/defense-of-dot-js/blob/master/proposal.md#typical-usage
    ...
}
```

Now you can publish it as an npm package and find it at unpkg.com to distribute it to your users.

?> incremental builds

?> profile

?> analyse tool

?> dirty chunks ([chunkhash])

?> source maps

?> PrefetchPlugin

?> resolving

?> DllPlugin

Development Tools

> see also resolving

webpack is not the only module bundler out there. If you are choosing between using webpack or any of the bundlers below, here is a feature-by-feature comparison on how webpack fares against the current competition.

Feature	webpack/webpack	jrburke/requirejs	substack browse
Additional chunks are loaded on demand	**yes**	**yes**	no
AMD `define`	**yes**	**yes**	deamdify
AMD `require`	**yes**	**yes**	no
AMD `require` loads on demand	**yes**	with manual configuration	no
CommonJS `exports`	**yes**	only wrapping in `define`	**yes**
CommonJS `require`	**yes**	only wrapping in `define`	**yes**
CommonJS `require.resolve`	**yes**	no	no
Concat in require `require("./fi" + "le")`	**yes**	no♦	no
Debugging support	**SourceUrl, SourceMaps**	not required	SourceMa
Dependencies	19MB / 127 packages	11MB / 118 packages	**1.2MB / 1 package**
ES2015 `import` / `export`	**yes** (webpack 2)	no	no
Expressions in require (guided) `require("./templates/" + template)`	**yes (all files matching included)**	no♦	no
Expressions in require (free) `require(moduleName)`	with manual configuration	no♦	no
Generate a single bundle	**yes**	yes♦	yes

Indirect require `var r = require; r("./file")`	**yes**	no♦	no
Load each file separate	no	yes	no
Mangle path names	**yes**	no	partial
Minimizing	uglify	uglify, closure compiler	uglifyify
Multi pages build with common bundle	with manual configuration	**yes**	with manu configurat
Multiple bundles	**yes**	with manual configuration	with manu configurat
Node.js built-in libs `require("path")`	**yes**	no	**yes**
Other Node.js stuff	process, __dir/filename, global	-	process, __dir/filen global
Plugins	**yes**	yes	**yes**
Preprocessing	**loaders, transforms**	loaders	transforms
Replacement for browser	`web_modules`, `.web.js`, package.json field, alias config option	alias option	package.j field, alias option
Requirable files	file system	**web**	file systen
Runtime overhead	**243B + 20B per module + 4B per dependency**	14.7kB + 0B per module + (3B + X) per dependency	415B + 25 module + 2X) per dependen

Watch mode	yes	not required	yes

♦ in production mode (opposite in development mode)

X is the length of the path string

?> require.main

?> require.cache

?> module.loaded

?> global

?> process

?> __dirname

?> __filename

?> module.id

As explained in detail on the concept page, Hot Module Replacement (HMR) exchanges, adds, or removes modules while an application is running without a page reload. HMR is particularly useful in applications using a single state tree, since components are "dumb" and will reflect the latest application state, even after their source is changed and they are replaced.

The approach described below uses Babel and React, but these tools are not necessary for HMR to work.

T> If you'd like to see examples of other approaches, please request them or better yet, open up a PR with an addition.

Project Config

This guide will be demonstrating the use of HMR with Babel, React, and PostCSS (using CSS Modules). To follow along, please add the following dependencies to your `package.json`:

To use HMR, you'll need the following dependencies:

```
npm install --save-dev babel-core@6.13.2 babel-loader@6.2.4 babel-preset-es2015@6.
13.2 babel-preset-react@6.11.1 babel-preset-stage-2@6.13.0 css-loader@0.23.1 postc
ss-loader@0.9.1 react-hot-loader@3.0.0-beta.6 style-loader@0.13.1 webpack@2.1.0-be
ta.25 webpack-dev-server@2.1.0-beta.0
```

In addition, for the purposes of this walkthrough, you'll need:

```
npm install --save react@15.3.0 react-dom@15.3.0
```

Babel Config

Your `.babelrc` file should look like the following:

```
{
  "presets": [
    ["es2015", {"modules": false}],
    // webpack understands the native import syntax, and uses it for tree shaking

    "stage-2",
    // Specifies what level of language features to activate.
    // Stage 2 is "draft", 4 is finished, 0 is strawman.
    // See https://tc39.github.io/process-document/
```

```
    "react"
    // Transpile React components to JavaScript
  ],
  "plugins": [
    "react-hot-loader/babel"
    // Enables React code to work with HMR.
  ]
}
```

webpack Config

While there're many ways of setting up your webpack config - via API, via multiple or single config files, etc - here is the basic information you should have available.

```
const { resolve } = require('path');
const webpack = require('webpack');

module.exports = {
  entry: [
    'react-hot-loader/patch',
    // activate HMR for React

    'webpack-dev-server/client?http://localhost:8080',
    // bundle the client for webpack-dev-server
    // and connect to the provided endpoint

    'webpack/hot/only-dev-server',
    // bundle the client for hot reloading
    // only- means to only hot reload for successful updates


    './index.js'
    // the entry point of our app
  ],
  output: {
    filename: 'bundle.js',
    // the output bundle

    path: resolve(__dirname, 'dist'),

    publicPath: '/'
    // necessary for HMR to know where to load the hot update chunks
  },

  context: resolve(__dirname, 'src'),

  devtool: 'inline-source-map',
```

```
  devServer: {
    hot: true,
    // enable HMR on the server

    contentBase: resolve(__dirname, 'dist'),
    // match the output path

    publicPath: '/'
    // match the output `publicPath`
  },

  module: {
    rules: [
      {
        test: /\.js$/,
        use: [
          'babel-loader',
        ],
        exclude: /node_modules/
      },
      {
        test: /\.css$/,
        use: [
          'style-loader',
          'css-loader?modules',
          'postcss-loader',
        ],
      },
    ],
  },

  plugins: [
    new webpack.HotModuleReplacementPlugin(),
    // enable HMR globally

    new webpack.NamedModulesPlugin(),
    // prints more readable module names in the browser console on HMR updates
  ],
};
```

There's a lot going on above, and not all of it is related to HMR. You may benefit from reading the webpack-dev-server options and the concept pages.

The basic assumption here is that your JavaScript entry is located at `./src/index.js` and that you're using CSS Modules for your styling.

Please see the comments inline that explain each portion of the config. The main areas to look are the `devServer` key and the `entry` key. The `HotModuleReplacementPlugin` is also necessary to include in the `plugins` array.

There are two modules included here for the purposes of this guide:

- The `react-hot-loader` addition to the entry, as noted above, is necessary to enable HMR with React components.
- The `NamedModulesPlugin` is a useful addition to better understand what modules are being updated when using HMR.

Code

In this guide, we're using the following files:

```
// ./src/index.js
import React from 'react';
import ReactDOM from 'react-dom';

import { AppContainer } from 'react-hot-loader';
// AppContainer is a necessary wrapper component for HMR

import App from './components/App';

const render = (Component) => {
  ReactDOM.render(
    <AppContainer>
      <Component/>
    </AppContainer>,
    document.getElementById('root')
  );
};

render(App);

// Hot Module Replacement API
if (module.hot) {
  module.hot.accept('./components/App', () => {
    render(App)
  });
}
```

```
// ./src/components/App.js
import React from 'react';
import styles from './App.css';

const App = () => (
  <div className={styles.app}>
    <h2>Hello, </h2>
  </div>
```

```
);

export default App;
```

```
// ./src/components/App.css
.app {
    text-size-adjust: none;
    font-family: helvetica, arial, sans-serif;
    line-height: 200%;
    padding: 6px 20px 30px;
}
```

The important thing to note in the code above is the `module` reference.

1. webpack will expose `module.hot` to our code since we set `devServer: { hot: true }` ;
2. Thus we can use the `module.hot` hook to enable HMR for specific resources (Here's `App.js`). The most important API here is `module.hot.accept` , which specifies how to handle changes to specific dependencies.
3. Note that because webpack 2 has built-in support for ES2015 modules, you won't need to re-require your root component in `module.hot.accept` . To make this work, you need to change the Babel ES2015 preset in `.babelrc` to be:

   ```
   ["es2015", {"modules": false}]
   ```

   like what we did in Babel Config. Note that disabling Babel's module plugin is not only necessary for HMR. If you don't disable it you'll run into many other issues (see Migrating from v1 to v2 and webpack-tree-shaking).
4. Note that if you're using ES2015 modules in your webpack 2 configuration file, and you change your `.babelrc` file in #3 above, you either need to use `require` or create two `.babelrc` files (issue here):
   - One in the project root directory with `"presets": ["es2015"]`
   - One in the home directory for webpack to build. For this example, in `src/` .

So in this case, `module.hot.accept` will fire the `render` method whenever `src/components/App.js` or its dependencies are changed - which means the `render` method will also fire when the `App.css` is changed, since `App.css` is included in `App.js` .

index.html

This needs to be placed inside of `dist` in your project root. webpack-dev-server will not run without it.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Example Index</title>
</head>
<body>
  <div id="root"></div>
  <script src="bundle.js"></script>
</body>
</html>
```

Package.json

Finally, we need to start up `webpack-dev-server` to bundle our code and see HMR in action. We can use the following `package.json` entry:

```
{
  "scripts" : {
    "start" : "webpack-dev-server"
  }
}
```

Run `npm start`, open up your browser to `http://localhost:8080`, and you should see the following entries printed in your console.log:

```
dev-server.js:49[HMR] Waiting for update signal from WDS…
only-dev-server.js:74[HMR] Waiting for update signal from WDS…
client?c7c8:24 [WDS] Hot Module Replacement enabled.
```

Go ahead and edit and save your `App.js` file. You should see something like the following in your console.log:

```
[WDS] App updated. Recompiling…
client?c7c8:91 [WDS] App hot update…
dev-server.js:45 [HMR] Checking for updates on the server…
log-apply-result.js:20 [HMR] Updated modules:
log-apply-result.js:22 [HMR]  - ./components/App.js
dev-server.js:27 [HMR] App is up to date.
```

Note that HMR specifies the paths of the updated modules. That's because we're using `NamedModulesPlugin`.

A component can lazily load dependencies without its consumer knowing using higher order functions, or a consumer can lazily load its children without its children knowing using a component that takes a function and collection of modules, or some combination of both.

LazilyLoad Component

Let's have a look at a consumer choosing to lazily load some components. The `importLazy` is simply a function that returns the `default` property, this is for Babel/ES2015 interoperability. If you don't need that you can omit the `importLazy` helper. The `importLazy` function simply returns whatever was exported as `export default` in the target module.

```
<LazilyLoad modules={{
  TodoHandler: () => importLazy(import('./components/TodoHandler')),
  TodoMenuHandler: () => importLazy(import('./components/TodoMenuHandler')),
  TodoMenu: () => importLazy(import('./components/TodoMenu')),
}}>
{({TodoHandler, TodoMenuHandler, TodoMenu}) => (
  <TodoHandler>
    <TodoMenuHandler>
      <TodoMenu />
    </TodoMenuHandler>
  </TodoHandler>
)}
</LazilyLoad>
```

Higher Order Component

As a component, you can also make sure your own dependencies are lazily loaded. This is useful if a component relies on a really heavy library. Let's say we have a Todo component that optionally supports code highlighting...

```
class Todo extends React.Component {
  render() {
    return (
      <div>
        {this.props.isCode ? <Highlight>{content}</Highlight> : content}
      </div>
    );
  }
}
```

We could then make sure the expensive library powering the Highlight component is only loaded when we actually want to highlight some code:

```
// Highlight.js
class Highlight extends React.Component {
  render() {
    const {Highlight} = this.props.highlight;
    // highlight js is now on our props for use
  }
}
export LazilyLoadFactory(Highlight, {
  highlight: () => import('highlight'),
});
```

Notice how the consumer of Highlight component had no idea it had a dependency that was lazily loaded? Or that if a user had todos with no code we would never need to load highlight.js?

The Code

Source code of the LazilyLoad component module which exports both the Component interface and the higher order component interface, as well as the importLazy function to make ES2015 defaults feel a bit more natural.

```
import React from 'react';

class LazilyLoad extends React.Component {

  constructor() {
    super(...arguments);
    this.state = {
      isLoaded: false,
    };
  }

  componentDidMount() {
    this._isMounted = true;
    this.load();
  }

  componentDidUpdate(previous) {
    if (this.props.modules === previous.modules) return null;
    this.load();
  }
```

```
  componentWillUnmount() {
    this._isMounted = false;
  }

  load() {
    this.setState({
      isLoaded: false,
    });

    const { modules } = this.props;
    const keys = Object.keys(modules);

    Promise.all(keys.map((key) => modules[key]()))
      .then((values) => (keys.reduce((agg, key, index) => {
        agg[key] = values[index];
        return agg;
      }, {})))
      .then((result) => {
        if (!this._isMounted) return null;
        this.setState({ modules: result, isLoaded: true });
      });
  }

  render() {
    if (!this.state.isLoaded) return null;
    return React.Children.only(this.props.children(this.state.modules));
  }
}

LazilyLoad.propTypes = {
  children: React.PropTypes.func.isRequired,
};

export const LazilyLoadFactory = (Component, modules) => {
  return (props) => (
    <LazilyLoad modules={modules}>
      {(mods) => <Component {...mods} {...props} />}
    </LazilyLoad>
  );
};

export const importLazy = (promise) => (
  promise.then((result) => result.default)
);

export default LazilyLoad;
```

Tips

- By using the bundle loader we can semantically name chunks to intelligently load groups of code.
- Make sure if you are using the babel-preset-2015, to turn modules to false, this will allow webpack to handle modules.

Dependencies

- ES2015 + JSX

References

- Higher Order Components
- react-modules
- Function as Child Components
- Example Repository
- Bundle Loader

webpack has a highly useful configuration that let you specify the base path for all the assets on your application. It's called `publicPath`.

Use cases

There are a few use cases on real applications where this feature becomes especially neat.

Set value on build time

For development mode what we usually have is an `assets/` folder that lives on the same level of our index page. This is fine but let's say you want to host all these static assets on a CDN on your production environment?

To approach this problem you can easily use a good old environment variable. Let's say we have a variable `ASSET_PATH`:

```
import webpack from 'webpack';

// Whatever comes as an environment variable, otherwise use root
const ASSET_PATH = process.env.ASSET_PATH || '/';

export default {
  output: {
    publicPath: ASSET_PATH
  },

  plugins: [
    // This makes it possible for us to safely use env vars on our code
    new webpack.DefinePlugin({
      'process.env.ASSET_PATH': JSON.stringify(ASSET_PATH)
    })
  ]
};
```

Set value on the fly

Another possible use case is to set the public path on the fly. webpack exposes a global variable that let's you do that, it's called `__webpack_public_path__`. So in your application entry point, you can simply do this:

```
__webpack_public_path__ = process.env.ASSET_PATH;
```

That's all you need. Since we're already using the `DefinePlugin` on our configuration, `process.env.ASSET_PATH` will always be defined so we can safely do that.

webpack is a module bundler, like Browserify or Brunch. It is not a task runner. Make, Grunt, or Gulp are task runners. But people get confused about the difference, so let's clear that up right away.

Task runners handle automation of common development tasks such as linting, building, or testing your project. Compared to bundlers, task runners have a higher level focus.

Bundlers help you get your JavaScript and stylesheets ready for deployment, transforming them into a format that's suitable for the browser. For example, JavaScript can be minified or split into chunks and loaded on-demand to improve performance. Bundling is one of the most important challenges in web development, and solving it well can remove a lot of pain from the process.

webpack can work alongside task runners. You can still benefit from their higher level tooling while leaving the problem of bundling to webpack. grunt-webpack and gulp-webpack are good examples.

T> Often webpack users use npm `scripts` as their task runner. This is a good starting point. Cross-platform support can become a problem, but there are several workarounds for that.

T> Even though webpack core focuses on bundling, you can find a variety of extensions that allow you to use it in a task runner kind of way.

?> Grunt

?> Gulp

?> Mocha

?> Karma

Tree shaking is a term commonly used in the JavaScript context for dead-code elimination, or more precisely, live-code import. It relies on ES2015 module import/export for the static structure of its module system. The name and concept have been popularized by the ES2015 module bundler rollup.

webpack 2 comes with a built-in support for ES2015 modules (alias *harmony modules*) as well as unused module export detection.

Example

Consider a **maths.js** library file exporting two functions, `square` and `cube`:

```
// This function isn't used anywhere
export function square(x) {
    return x * x;
}

// This function gets included
export function cube(x) {
    return x * x * x;
}
```

In our **main.js** we are selectively importing `cube`:

```
import {cube} from './maths.js';
console.log(cube(5)); // 125
```

Running `node_modules/.bin/webpack main.js dist.js` and inspecting `dist.js` reveals that `square` is not being exported (see the "unused harmony export square" comment):

```
/* ... webpackBootstrap ... */
/******/ ([
/* 0 */
/***/ (function(module, __webpack_exports__, __webpack_require__) {

"use strict";
/* unused harmony export square */
/* harmony export (immutable) */ __webpack_exports__["a"] = cube;
// This function isn't used anywhere
function square(x) {
  return x * x;
}

// This function gets included
```

```
function cube(x) {
  return x * x * x;
}

/***/ }),
/* 1 */
/***/ (function(module, __webpack_exports__, __webpack_require__) {

"use strict";
Object.defineProperty(__webpack_exports__, "__esModule", { value: true });
/* harmony import */ var __WEBPACK_IMPORTED_MODULE_0__maths_js__ = __webpack_requi
re__(0);

console.log(__webpack_require__.i(__WEBPACK_IMPORTED_MODULE_0__maths_js__["a" /* c
ube */])(5)); // 125

/***/ })
```

When running a production build, `node_modules/.bin/webpack --optimize-minimize main.js dist.min.js`, only the minimized version of `cube` but not `square` remains in the build:

```
/* ... */
function(e,t,n){"use strict";function r(e){return e*e*e}t.a=r}
/* ... */
function(e,t,n){"use strict";Object.defineProperty(t,"__esModule",{value:!0});var
r=n(0);console.log(n.i(r.a)(5))}
```

Weblinks

- Tree-shaking with webpack 2 and Babel 6
- Webpack 2 Tree Shaking Configuration

TypeScript is a typed superset of JavaScript that compiles to plain JavaScript, in this guide we will learn how to integrate Typescript with webpack.

Basic Setup

In order to get started with webpack and Typescript, first we must install webpack in our project. If you didn't do so already please check out webpack Installation Guide.

To start using webpack with Typescript you need a couple of things:

1. Install the Typescript compiler in your project.
2. Install a Typescript loader (in this case we're using ts-loader).
3. Create a **tsconfig.json** file to contain our TypeScript compilation configuration.
4. Create **webpack.config.js** to contain our webpack configuration.

You can install the TypeScript compiler and the TypeScript loader from npm by running:
`npm install --save-dev typescript ts-loader`

tsconfig.json

The tsconfig file can start as an empty configuration file, here you can see an example of a basic configuration for TypeScript to compile to es5 as well as providing support for JSX.

```
{
  "compilerOptions": {
    "outDir": "./dist/",
    "sourceMap": true,
    "noImplicitAny": true,
    "module": "commonjs",
    "target": "es5",
    "jsx": "react",
    "allowJs": true
  }
}
```

You can read more about tsconfig.json configuration options at the TypeScript documentation website

webpack.config.js

A basic webpack with TypeScript config should look along these lines:

```
module.exports = {
```

```
  entry: './index.ts',
  output: {
    filename: '/bundle.js',
    path: '/'
  },
  module: {
    rules: [
      {
        test: /\.tsx?$/,
        loader: 'ts-loader',
        exclude: /node_modules/,
      },
    ]
  },
  resolve: {
    extensions: [".tsx", ".ts", ".js"]
  },
};
```

Here we specify our entry point to be **index.ts** in our current directory, an output file called **bundle.js** and our TypeScript loader that is in charge of compiling our TypeScript file to JavaScript. We also add `resolve.extensions` to instruct webpack what file extensions to use when resolving Typescript modules.

Typescript loaders

Currently there are 2 loaders for TypeScript available:

- `awesome-typescript-loader`
- `ts-loader`

Awesome TypeScript loader has created a wonderful explanation of the difference between `awesome-typescript-loader` and `ts-loader`.

You can read more about it here.

In this guide we will be using `ts-loader` as currently it is easier enabling additional webpack features such as importing non code assets into your project.

Enabling source maps

In order to enable source maps we first must configure TypeScript to output inline source maps to our compiled JavaScript files. This is done by setting the sourceMap property to true.

tsconfig.json

```
{
  "sourceMap": true
}
```

Once TypeScript is configured to output source maps we need to tell webpack to extract these source maps and pass them to the browser, this way we will get the source file exactly as we see it in our code editor.

webpack.config.js

```
module.exports = {
 entry: './index.ts',
 output: {
   filename: '/bundle.js',
   path: '/'
 },
 module: {
   rules: [
     {
       enforce: 'pre',
       test: /\.js$/,
       loader: "source-map-loader"
     },
     {
       enforce: 'pre',
       test: /\.tsx?$/,
       use: "source-map-loader"
     }
   ]
 },
 resolve: {
   extensions: [".tsx", ".ts", ".js"]
 },
 devtool: 'inline-source-map',
};
```

First we add a new loader called `source-map-loader`.

To install it run:

`npm install --save-dev source-map-loader`.

Once the loader is installed we need to tell webpack we want to run this loader before any other loaders by using the `enforce: 'pre'` configuration flag. Finally we need to enable source maps in webpack by specifying the `devtool` property. Currently we use the 'inline-source-map' setting, to read more about this setting and see other options check out the devtool documentation.

Using 3rd Party Libraries

When installing 3rd party libraries from npm, it is important to remember to install the typing definition for that library.

You can install 3rd party library definitions from the @types repository.

For example if we want to install lodash we can run the following command to get the typings for it: `npm install --save-dev @types/lodash`

For more information see this blog post

Importing non code assets

To use non code assets with TypeScript, we need to tell TypeScript how to defer the type for these imports.

To do this we need to create a **custom.d.ts** file. This file signifies custom definitions for TypeScript in our project.

In our **custom.d.ts** file we need to provide a definition for svg imports, to do this we need to put the following content in this file:

```
declare module "*.svg" {
  const content: any;
  export default content;
}
```

Here we declare a new module for svg by specifying any import that ends in **.svg** and define the type for this module as any. If we wanted to be more explicit about this being a url we could define the type as string.

This applies not only to svg but any custom loader you may want to use which includes css, scss, json or any other file you may wish to load in your project.

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

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

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

Options

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

module.exports = {
  // click on the name of the option to get to the detailed documentation
  // click on the items with arrows to show more examples / advanced options

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

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

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

    <details><summary>[filename](/configuration/output#output-filename): "bundle.j
s", // string</summary>
    [filename](/configuration/output#output-filename): "[name].js", // for multipl
e entry points
    [filename](/configuration/output#output-filename): "[chunkhash].js", // for [l
ong term caching](/guides/caching)
    </details>
```

```
    // the filename template for entry chunks

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

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

    <details><summary>[libraryTarget](/configuration/output#output-librarytarget):
 "umd", // universal module definition</summary>
        [libraryTarget](/configuration/output#output-librarytarget): "umd2", // un
iversal module definition
        [libraryTarget](/configuration/output#output-librarytarget): "commonjs2",
// exported with module.exports
        [libraryTarget](/configuration/output#output-librarytarget): "commonjs-mod
ule", // exports with module.exports
        [libraryTarget](/configuration/output#output-librarytarget): "commonjs", /
/ exported as properties to exports
        [libraryTarget](/configuration/output#output-librarytarget): "amd", // def
ined with AMD defined method
        [libraryTarget](/configuration/output#output-librarytarget): "this", // pr
operty set on this
        [libraryTarget](/configuration/output#output-librarytarget): "var", // var
iable defined in root scope
        [libraryTarget](/configuration/output#output-librarytarget): "assign", //
blind assignment
        [libraryTarget](/configuration/output#output-librarytarget): "window", //
property set to window object
        [libraryTarget](/configuration/output#output-librarytarget): "global", //
property set to global object
        [libraryTarget](/configuration/output#output-librarytarget): "jsonp", // j
sonp wrapper
    </details>
    // the type of the exported library

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
          }
        ]
      },

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

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

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

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

      { [resource](/configuration/module#rule-resource): { [not](/configuration/mo
dule#condition): /* condition */ } }
      // matches if the condition is not matched
    ],

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

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

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

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

    [modules](/configuration/resolve#resolve-modules): [
```

```
      "node_modules",
      path.resolve(__dirname, "app")
    ],
    // directories where to look for modules

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

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

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

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

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

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

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

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

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

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

    [mainFields](/configuration/resolve#resolve-mainfields): ["main"],
```

```
    // properties that are read from description file
    // when a folder is requested

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

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

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

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

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

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

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

```
  <details><summary>[devtool](/configuration/devtool): "source-map", // enum </sum
mary>
  [devtool](/configuration/devtool): "inline-source-map", // inlines SourceMap int
o orginal file
  [devtool](/configuration/devtool): "eval-source-map", // inlines SourceMap per m
odule
  [devtool](/configuration/devtool): "hidden-source-map", // SourceMap without ref
erence in original file
  [devtool](/configuration/devtool): "cheap-source-map", // cheap-variant of Sourc
eMap without module mappings
  [devtool](/configuration/devtool): "cheap-module-source-map", // cheap-variant o
f SourceMap with module mappings
  [devtool](/configuration/devtool): "eval", // no SourceMap, but named modules. F
astest at the expense of detail.
  </details>
  // enhance debugging by adding meta info for the browser devtools
  // source-map most detailed at the expense of build speed.

  [context](/configuration/entry-context#context): __dirname, // string (absolute
path!)
  // the home directory for webpack
  // the [entry](/configuration/entry-context) and [module.rules.loader](/configur
ation/module#rule-loader) option
  //   is resolved relative to this directory

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

  <details><summary>[externals](/configuration/externals): ["react", /^@angular\//
],</summary>
  [externals](/configuration/externals): "react", // string (exact match)
  [externals](/configuration/externals): /^[a-z\-]+($|\/)/, // Regex
  [externals](/configuration/externals): { // object
    angular: "this angular", // this["angular"]
    react: { // UMD
      commonjs: "react",
      commonjs2: "react",
      amd: "react",
      root: "React"
    }
  },
```

```
  [externals](/configuration/externals): (request) => { /* ... */ return "commonjs
 " + request }
  </details>
  // Don't follow/bundle these modules, but request them at runtime from the envir
onment

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

  [devServer](/configuration/dev-server): {
    /* TODO */
  },

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


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

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

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

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

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

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

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

    [poll](watch#watchoptions-poll): true,
    [poll](watch#watchoptions-poll): 500, // intervall in ms
    // enables polling mode for watching
```

```
    // must be used on filesystems that doesn't notify on change
    // i. e. nfs shares
  },

  [node](node): {
    /* TODO */
  },

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

  </details>
}
```

...

?> exporting a function and --env

?> returning a Promise

?> exporting multiple configurations

webpack lets you define you configuration files in any language. The list of supported file extensions can be found at the node-interpret package. webpack with the help of node-interpret will run your configuration through the language of your choice.

For example if you use **coffeescript**, your file would be as follows:

webpack.config.coffee

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

?> loader settings, e.g. `sassLoader`, `eslint` sections

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

`context`

`string`

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

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

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

`entry`

`string | [string] | object { <key>: string | [string] }`

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

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

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

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

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

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

`output.chunkFilename`

`string`

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

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

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

`output.crossOriginLoading`

`boolean` `string`

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

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

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

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

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

`output.devtoolFallbackModuleFilenameTemplate`

`string | function(info)`

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

See `output.devtoolModuleFilenameTemplate`.

`output.devtoolLineToLine`

`boolean | object`

(Deprecated: Not really used, not really usable, write an issue if you have a other opinion)

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

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

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

`output.devtoolModuleFilenameTemplate`

`string | function(info)`

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

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

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

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

Template	Description
[absolute-resource-path]	The absolute filename
[all-loaders]	Automatic and explicit loaders and params up to the name of the first loader
[hash]	The hash of the module identifier

[id]	The module identifier
[loaders]	Explicit loaders and params up to the name of the first loader
[resource]	The path used to resolve the file and any query params used on the first loader
[resource-path]	The path used to resolve the file without any query params

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

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

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

`output.filename`

`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.

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

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

Using entry name:

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

Using internal chunk id:

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

Using the unique hash generated for every build:

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

Using hashes based on each chunks' content:

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

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

The default value is `"[name].js"`.

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

Note this options does not affect output files for on-demand-loaded chunks. For these files the `output.chunkFilename` option is used. It also doesn't affect files created by loaders. For these files see loader options.

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

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

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

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

`output.hotUpdateChunkFilename`

`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:

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

Here is no need to change it.

`output.hotUpdateFunction`

`function`

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

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

For details see `output.jsonpFunction`.

`output.hotUpdateMainFilename`

`string`

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

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

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

Here is no need to change it.

`output.jsonpFunction`

`function`

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

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

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

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

`output.library`

`string`

Read the library guide for details.

Use `library`, and `libraryTarget` below, when writing a JavaScript library that should export values, which can be used by other code depending on it. Pass a string with the name of the library:

```
library: "MyLibrary"
```

The name is used depending on the value of the `output.libraryTarget` options.

Note that `output.libraryTarget` defaults to `var`. This means if only `output.library` is used it is exported as variable declaration (when used as script tag it's available in the global scope after execution).

`output.libraryTarget`

`string`

> Default: `"var"`

Read the library guide for details.

Configure how the library will be exposed. Any one of the following options can be used.

> To give your library a name, set the `output.library` config to it (the examples assume `library: "MyLibrary"`)

The following options are supported:

`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_;

// your users will use your library like:
MyLibrary.doSomething();
```

(Not specifying a `output.library` will cancel this var configuration)

`libraryTarget: "this"` - When your library is loaded, the **return value of your entry point** will be assigned to this, the meaning of `this` is up to you:

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

// your users will use your library like:
this.MyLibrary.doSomething();
MyLibrary.doSomething(); //if this is window
```

`libraryTarget: "window"` - When your library is loaded, **the return value of your entry point** will be part `window` object.

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

//your users will use your library like:
window.MyLibrary.doSomething();
```

`libraryTarget: "global"` - When your library is loaded, **the return value of your entry point** will be part `global` object.

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

//your users will use your library like:
global.MyLibrary.doSomething();
```

`libraryTarget: "commonjs"` - When your library is loaded, **the return value of your entry point** will be part of the exports object. As the name implies, this is used in CommonJS environments:

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

//your users will use your library like:
require("MyLibrary").doSomething();
```

`libraryTarget: "commonjs2"` - When your library is loaded, **the return value of your entry point** will be part of the exports object. As the name implies, this is used in CommonJS environments:

```
module.exports = _entry_return_;

//your users will use your library like:
require("MyLibrary").doSomething();
```

Wondering the difference between CommonJS and CommonJS2? Check this out (they are pretty much the same).

`libraryTarget: "commonjs-module"` - Expose it using the `module.exports` object (`output.library` is ignored), `__esModule` is defined (it's threaded as ES2015 Module in interop mode)

`libraryTarget: "amd"` - In this case webpack will make your library an AMD module.

But there is a very important pre-requisite, your entry chunk must be defined with the define property, if not, webpack will create the AMD module, but without dependencies. The output will be something like this:

```
define([], function() {
    //what this module returns is what your entry chunk returns
});
```

But if you download this script, first you may get a error: `define is not defined` , it's ok! If you are distributing your library with AMD, then your users need to use RequireJS to load it.

Now that you have RequireJS loaded, you can load your library.

But, `require([ _what?_ ])` ?

`output.library` !

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

So your module will be like:

```
define("MyLibrary", [], function() {
    //what this module returns is what your entry chunk returns
});
```

And you can use it like this:

```
// And then your users will be able to do:
require(["MyLibrary"], function(MyLibrary){
    MyLibrary.doSomething();
});
```

`libraryTarget: "umd"` - This is a way for your library to work with all the module definitions (and where aren't modules at all). It will work with CommonJS, AMD and as global variable. You can check the UMD Repository to know more about it.

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

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

And finally the output is:

```
(function webpackUniversalModuleDefinition(root, factory) {
    if(typeof exports === 'object' && typeof module === 'object')
        module.exports = factory();
    else if(typeof define === 'function' && define.amd)
        define("MyLibrary", [], factory);
    else if(typeof exports === 'object')
        exports["MyLibrary"] = factory();
    else
        root["MyLibrary"] = factory();
})(this, function() {
    //what this module returns is what your entry chunk returns
});
```

Module proof library.

`libraryTarget: "assign"` - Here webpack will blindly generate an implied global.

```
 MyLibrary = _entry_return_;
```

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

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

```
  MyLibrary( _entry_return_ );
```

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

`output.path`

`string`

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

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

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

`output.pathinfo`

`boolean`

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

```
pathinfo: true
```

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

`output.publicPath`

`string`

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

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

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

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

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

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

For this configuration:

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

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

A loader outputting HTML might emit something like this:

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

or when loading an image in CSS:

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

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

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

Examples:

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

```
publicPath: "../assets/", // relative to HTML page
publicPath: "", // relative to HTML page (same directory)
```

output.sourceMapFilename

`string`

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

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

Technically the `[name]`, `[id]`, `[hash]` and `[chunkhash]` placeholders can be used, if generating a SourceMap for chunks. In addition to that the `[file]` placeholder is replaced with the filename of the original file. It's recommended to only use the `[file]` placeholder, as the other placeholders won't work when generating SourceMaps for non-chunk files. Best leave the default.

output.sourcePrefix

`string`

Change the prefix for each line in the output bundles.

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

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

There is no need to change it.

output.umdNamedDefine

`boolean`

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

```
umdNamedDefine: true
```

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

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

`module.noParse`

`RegExp | [RegExp]`

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

```
noParse: /jquery|lodash/
```

`module.rules`

`array`

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

Rule

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

Rule conditions

There are two input values for the conditions:

1. The resource: An absolute path to the file requested. It's already resolved according 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"` from `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.

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 affect the loader category. Whether it's an normal, pre- or post-loader.

The `parser` property affect the parser options.

Nested rules

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

These rules are evaluated when the Rule condition matches.

`Rule.enforce`

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

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

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

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

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

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

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

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

Rule.exclude

`Rule.exclude` is a shortcut to `Rule.resource.exclude`. See `Rule.resource` and `Condition.exclude` for details.

Rule.include

`Rule.include` is a shortcut to `Rule.resource.include`. See `Rule.resource` and `Condition.include` for details.

Rule.issuer

A `Condition` matched with the issuer. See details in `Rule` conditions.

Rule.loader

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

Rule.loaders

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

It exists for compatibility reasons. Use `Rule.use` instead.

Rule.oneOf

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

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` only exists for compatibility reasons. Use `Rule.options` instead.

Rule.parser

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

For each different parser options object a new parser is created and plugins can apply plugins depending on the parser options. Many of the default plugins apply their parser plugins only if a property in the parser options is not set or true.

Examples (parser options by the default plugins):

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

Rule.resource

A `Condition` matched with the resource. See details in `Rule` conditions.

Rule.rules

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

Rule.test

`Rule.test` is a shortcut to `Rule.resource.test`. See `Rule.resource` and `Condition.test` for details.

Rule.use

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

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

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

```
use: [
  {
    loader: 'style-loader'
  },
  {
    loader: 'css-loader',
    options: {
      importLoaders: 1
    }
  },
  {
    loader: 'less-loader',
    options: {
      noIeCompat: true
    }
  }
]
```

See UseEntry for details.

Condition

Conditions can be one of these:

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

- A object: All properties must match. Each property has a defined behavior.

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

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

`{ exclude: Condition }` : The Condition must NOT match. The convention is the 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 }` : The Condition must NOT match.

Example:

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

UseEntry

`object`

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

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

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

Example:

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

```
    }
  }
```

Note that webpack need to generate an unique module identifier from 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%, but may be not unique if you apply the same loaders with different options to the same resource and the options have some stringified values. It also breaks if the options object cannot be stringified (i. e. circular JSON). Because of this you can have a `ident` property in the options object which is used as unique identifier.

Module Contexts

(Deprecated)

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

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

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

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

Here are the available options with their defaults:

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

Note: 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`

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

`resolve`

`object`

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

`resolve.alias`

`object`

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

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

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

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

you can use the alias:

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

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

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

which would yield these results:

```
import Test1 from 'xyz'; // Success, file.js is resolved and imported
import Test2 from 'xyz/file.js'; // Error, /path/to/file.js/file.js is invalid
```

The following table explains a lot more cases:

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

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

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

`resolve.aliasFields`

`string`

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

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

`resolve.descriptionFiles`

`array`

The JSON files to use for descriptions. Default:

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

`resolve.enforceExtension`

`boolean`

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

```
enforceExtension: false
```

`resolve.enforceModuleExtension`

`boolean`

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

```
enforceModuleExtension: false
```

`resolve.extensions`

`array`

Automatically resolve certain extensions. This defaults to:

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

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

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

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

`resolve.mainFields`

`array`

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

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

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

For any other target (including `node`):

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

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

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

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

`resolve.mainFiles`

`array`

The filename to be used while resolving directories. Default:

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

`resolve.modules`

`array`

Tell webpack what directories should be searched when resolving modules.

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

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

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

`resolve.modules` defaults to:

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

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

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

`resolve.unsafeCache`

`regex` `array` `boolean`

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

```
unsafeCache: true
```

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

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

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

`resolveLoader`

`object`

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

```
{
    modules: ["web_loaders", "web_modules", "node_loaders", "node_modules"],
    extensions: [".webpack-loader.js", ".web-loader.js", ".loader.js", ".js"],
    packageMains: ["webpackLoader", "webLoader", "loader", "main"]
}
```

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

`resolveLoader.moduleExtensions`

`array`

The extensions which are tried when resolving a module (e.g. loaders). By default this is an empty array.

If you want to use loaders without the `-loader` suffix, you can use this:

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

`resolve.plugins`

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

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

`resolve.symlinks`

`boolean`

Whether to resolve symlinks to their symlinked location. Default:

```
symlinks: true
```

`resolve.cachePredicate`

`function`

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

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

?> `plugins` customize the webpack build process in a variety of ways. This page discusses using existing plugins, however if you are interested in writing your own please visit Writing a Plugin.

`plugins`

`array`

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

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

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

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

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

```
    // webpack-dev-server enhancement plugins
    new DashboardPlugin(),
    new webpack.HotModuleReplacementPlugin(),
  ]
```

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

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

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

`devServer`

`object`

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

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

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

```
http://localhost:9000/
webpack result is served from /build/
content is served from 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.

`devServer.clientLogLevel`

`string`

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

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

```
clientLogLevel: "none"
```

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

Note that the console will *always* show bundle errors and warnings. This option only effects the message before it.

devServer.compress

`boolean`

Enable gzip compression for everything served:

```
compress: true
```

devServer.contentBase

`boolean` `string` `array`

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

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

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

Note that it is recommended to use an absolute path.

It is also possible to serve from multiple directories:

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

To disable `contentBase`:

```
contentBase: false
```

devServer.filename

`string`

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

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

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

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

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

devServer.headers

`object`

Adds headers to all requests:

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

devServer.historyApiFallback

`boolean` `object`

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

```
historyApiFallback: true
```

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

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

```
      { from: /^\/subpage/, to: '/views/subpage.html' },
      { from: /./, to: '/views/404.html' }
    ]
  }
```

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

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

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

`devServer.host` - CLI only

`string`

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

```
host: "0.0.0.0"
```

`devServer.hot`

`boolean`

Enable webpack's Hot Module Replacement feature:

```
hot: true
```

?> Add various other steps needed for this to work. (From my experience, and the current docs it looks like other steps are needed here - not like in the cmd line where it's just a flag)

`devServer.hotOnly` - CLI only

`boolean`

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

```
hotOnly: true
```

`devServer.https`

`boolean` `object`

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

```
https: true
```

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

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

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

`devServer.inline` - CLI only

`boolean`

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

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

```
inline: false
```

T> Inline mode is recommended when using Hot Module Replacement.

`devServer.lazy`

`boolean`

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

```
lazy: true
```

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

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

`devServer.noInfo`

`boolean`

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

```
noInfo: true
```

`devServer.port` - CLI only

`number`

Specify a port number to listen for requests on:

```
port: 8080
```

`devServer.proxy`

`object`

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

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

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

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

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

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

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

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

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

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

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

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

```
proxy: {
  "/api": {
    target: "http://localhost:3000",
    bypass: function(req, res, proxyOptions) {
```

```
      if (req.headers.accept.indexOf("html") !== -1) {
        console.log("Skipping proxy for browser request.");
        return "/index.html";
      }
    }
  }
}
```

`devServer.public` - CLI only

`string`

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

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

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

`devServer.publicPath`

`string`

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

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

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

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

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

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

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

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

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

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

devServer.quiet

`boolean`

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

```
quiet: true
```

devServer.setup

`function`

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

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

devServer.staticOptions

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

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

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

devServer.stats

`string` `object`

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

To show only errors in your bundle:

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

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

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

`devServer.watchContentBase`

`boolean`

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

```
watchContentBase: true
```

It is disabled by default.

`devServer.watchOptions`

`object`

Control options related to watching the files.

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

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

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

See WatchOptions for more options.

This option controls if and how Source Maps are generated.

`devtool`

`string` `false`

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

devtool	build	rebuild	production	quality
eval	+++	+++	no	generated code
cheap-eval-source-map	+	++	no	transformed code (lines only)
cheap-source-map	+	o	yes	transformed code (lines only)
cheap-module-eval-source-map	o	++	no	original source (lines only)
cheap-module-source-map	o	-	yes	original source (lines only)
eval-source-map	--	+	no	original source
source-map	--	--	yes	original source
nosources-source-map	--	--	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.

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

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.

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

`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.

`cheap-module-eval-source-map` - Like `eval-source-map`, each module is executed with `eval()` and a SourceMap is added as a DataUrl to the `eval()`. It is "cheap" because it doesn't have column mappings, it only maps line numbers.

For production

`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.

`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.

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

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

`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.

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

?> This page needs more information to make it easier for users to choose a good option.

References

- Enabling Sourcemaps
- webpack devtool source map

webpack can compile for multiple environments or *targets*. To understand what a target is in detail, read the concepts.

`target`

`string`

Tells webpack which environment the application is targeting. The following values are supported:

`target`	Description
`async-node`	Compile for usage in a Node.js-like environment (uses `fs` and `vm` to load chunks asynchronously)
`electron`	Alias for `electron-main`
`electron-main`	Compile for Electron for main process.
`electron-renderer`	Compile for Electron for renderer process, providing a target using `JsonpTemplatePlugin`, `FunctionModulePlugin` for browser environments and `NodeTargetPlugin` and `ExternalsPlugin` for CommonJS and Electron built-in modules.
`node`	Compile for usage in a Node.js-like environment (uses Node.js `require` to load chunks)
`node-webkit`	Compile for usage in WebKit and uses JSONP for chunk loading. Allows importing of built-in Node.js modules and `nw.gui` (experimental)
`web`	Compile for usage in a browser-like environment **(default)**
`webworker`	Compile as WebWorker

For example, when the `target` is set to `"electron"`, webpack includes multiple electron specific variables. For more information on which templates and externals are used, you can refer to webpack's source code.

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 properly for you.

`watch`

`boolean`

Turn on watch mode. This means that after the initial build, webpack will continue to watch for changes in any of the resolved files. Watch mode is turned off by default:

```
watch: false
```

T> In webpack-dev-server and webpack-dev-middleware watch mode is enabled by default.

`watchOptions`

`object`

A set of options used to customize watch mode:

```
watchOptions: {
  aggregateTimeout: 300,
  poll: 1000
}
```

`watchOptions.aggregateTimeout`

`number`

Add a delay before rebuilding once the first file changed. This allows webpack to aggregate any other changes made during this time period into one rebuild. Pass a value in milliseconds:

```
aggregateTimeout: 300 // The default
```

`watchOptions.ignored`

For some systems, watching many file systems can result in a lot of CPU or memory usage. It is possible to exclude a huge folder like `node_modules`:

```
ignored: /node_modules/
```

It is also possible to use anymatch patterns:

```
ignored: "files/**/*.js"
```

`watchOptions.poll`

`boolean` `number`

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

```
poll: 1000 // Check for changes every second
```

T> If watching does not work for you, try out this option. Watching does not work with NFS and machines in VirtualBox.

`externals` configuration in webpack provides a way of not including a dependency in the bundle. Instead the created bundle relies on that dependency to be present in the consumers environment. This typically applies to library developers though application developers can make good use of this feature too.

`externals`

`string` `regex` `function` `array` `object`

Prevent bundling of certain `import`ed packages and instead retrieve these *external packages at runtime*.

For example, to include jQuery from a CDN instead of bundling it:

index.html

```
...
<script src="https://code.jquery.com/jquery-3.1.0.js"
  integrity="sha256-slogkvB1K3VOkzAI8QITxV3VzpOnkeNVsKvtkYLMjfk="
  crossorigin="anonymous"></script>
...
```

webpack.config.js

```
externals: {
  jquery: 'jQuery'
}
```

This leaves any dependant modules unchanged, i.e. the code shown below will still work:

```
import $ from 'jquery';

$('.my-element').animate(...);
```

T> **consumer** here is any end user application that includes the library that you have bundled using webpack.

Your bundle which has external dependencies can be used in various module contexts mainly CommonJS, AMD, global and ES2015 modules. The external library may be available in any of the above form but under different variables.

`externals` supports the following module contexts

- **global** - An external library can be available as a global variable. The consumer can achieve this by including the external library in a script tag. This is the default setting for externals.
- **commonjs** - The consumer application may be using a CommonJS module system and hence the external library should be available as a CommonJS module.
- **commonjs2** - Similar to the above line but where the export is `module.exports.default`.
- **amd** - Similar to the above line but using AMD module system.

`externals` accepts various syntax and interprets them in different manners.

string

`jQuery` in the externals indicates that your bundle will need `jQuery` variable in the global form.

array

```
externals: {
  subtract: ['./math', 'subtract']
}
```

`subtract: ['./math', 'subtract']` converts to a parent child construct, where `./math` is the parent module and your bundle only requires the subset under `subtract` variable.

object

```
externals : {
  react: 'react'
}

// or

externals : {
  lodash : {
    commonjs: "lodash",
    amd: "lodash",
    root: "_" // indicates global variable
  }
}
```

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.

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 and provides some options to, for example, whitelist packages.

It basically comes down to this:

```
externals: [
  function(context, request, callback) {
    if (/^yourregex$/.test(request)){
      return callback(null, 'commonjs ' + request);
    }
    callback();
  }
],
```

The `'commonjs ' + request` defines the type of module that needs to be externalized.

regex

?> TODO - I think its overkill to list externals as regex.

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

node

`object`

Customize the NodeJS environment using polyfills or mocks:

```
node: {
  console: false,
  global: true,
  process: true,
  Buffer: true,
  __filename: "mock",
  __dirname: "mock",
  setImmediate: true
}
```

?> Elaborate on this section. What does "mock" or "empty" do? Does `<node builtin>` in the current documentation mean you can enable, disable, or polyfill any global Node.js function? (it seems `setImmediate` is the example for that)

These options allows you to control how webpack notifies you of assets and entrypoints that exceed a specific file limit. This feature was inspired by the idea of webpack Performance Budgets.

`performance`

`object`

Configure how performance hints are shown. For example if you have an asset that is over 250kb, webpack will emit a warning notifiying you of this.

`performance.hints`

`boolean | "error" | "warning"`

Turns hints on/off. In addition, tells webpack to throw either an error or a warning when hints are found. This property is set to `"warning"` by default.

Given an asset is created that is over 250kb:

```
performance: {
  hints: false
}
```

No hint warnings or errors are shown.

```
performance: {
  hints: "warning"
}
```

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

```
performance: {
  hints: "error"
}
```

An error will be displayed notifying you of a large asset. We recommend using `hints: "error"` during production builds to help prevent deploying production bundles that are too large, impacting webpage performance.

`performance.maxEntrypointSize`

`int`

An entrypoint represents all assets that would be utilized during initial load time for a specific entry. This option controls when webpack should emit performance hints based on the maximum entrypoint size. The default value is `250000` (bytes).

```
performance: {
  maxEntrypointSize: 400000
}
```

`performance.maxAssetSize`

`int`

An asset is any emitted file from webpack. This option controls when webpack emits a performance hint based on individual asset size. The default value is `250000` (bytes).

```
performance: {
  maxAssetSize: 100000
}
```

`performance.assetFilter`

`Function`

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

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

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

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

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

The `stats` option lets you precisely control what bundle information gets displayed. This can be a nice middle ground if you don't want to use `quiet` or `noInfo` because you want some bundle information, but not all of it.

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

W> This option does not have any effect when using the Node.js API.

`stats`

`object` `string`

There are some presets available to use as a shortcut. Use them like this:

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

Preset	Alternative	Description
`"errors-only"`	*none*	Only output when errors happen
`"minimal"`	*none*	Only output when errors or new compilation happen
`"none"`	`false`	Output nothing
`"normal"`	`true`	Standard output
`"verbose"`	*none*	Output everything

For more granular control, it is possible to specify exactly what information you want. Please note that all of the options in this object are optional.

```
stats: {
  // Add asset Information
  assets: true,
  // Sort assets by a field
  assetsSort: "field",
  // Add information about cached (not built) modules
  cached: true,
  // Add children information
  children: true,
  // Add chunk information (setting this to `false` allows for a less verbose outp
ut)
  chunks: true,
  // Add built modules information to chunk information
```

```
  chunkModules: true,
  // Add the origins of chunks and chunk merging info
  chunkOrigins: true,
  // Sort the chunks by a field
  chunksSort: "field",
  // Context directory for request shortening
  context: "../src/",
  // `webpack --colors` equivalent
  colors: true,
  // Add errors
  errors: true,
  // Add details to errors (like resolving log)
  errorDetails: true,
  // Add the hash of the compilation
  hash: true,
  // Add built modules information
  modules: true,
  // Sort the modules by a field
  modulesSort: "field",
  // Add public path information
  publicPath: true,
  // Add information about the reasons why modules are included
  reasons: true,
  // Add the source code of modules
  source: true,
  // Add timing information
  timings: true,
  // Add webpack version information
  version: true,
  // Add warnings
  warnings: true
};
```

?> These are all the other options that might not need an entire page. Either we need to create new pages for them, move them to an existing page, or keep an **Other Options** section like this and replace this TODO with a short description/lead-in paragraph.

`amd`

`object`

Set the value of `require.amd` or `define.amd`:

```
amd: {
  jQuery: true
}
```

Certain popular modules written for AMD, most notably jQuery versions 1.7.0 to 1.9.1, will only register as an AMD module if the loader indicates it has taken special allowances for multiple versions being included on a page. The allowances were the ability to restrict registrations to a specific version or to support different sandboxes with different defined modules.

This option allows you to set the key your module looks for to a truthy value. As it happens, the AMD support in webpack ignores the defined name anyways.

`bail`

`boolean`

Fail out on the first error instead of tolerating it. By default webpack will log these errors in red in the terminal, as well as the browser console when using HMR, but continue bundling. To enable it:

```
bail: true
```

This will force webpack to exit its bundling process.

`cache`

`boolean` `object`

Cache the generated webpack modules and chunks to improve build speed. Caching is enabled by default while in watch mode. To disable caching simply pass:

```
cache: false
```

If an object is passed, webpack will use this object for caching. Keeping a reference to this object will allow one to share the same cache between compiler calls:

```
let SharedCache = {};

export default {
  ...,
  cache: SharedCache
}
```

W> Don't share the cache between calls with different options.

?> Elaborate on the warning and example - calls with different configuration options?

`loader`

`object`

Expose custom values into the loader context.

?> Add an example...

`profile`

`boolean`

Capture a "profile" of the application, including statistics and hints, which can then be dissected using the Analyze tool.

T> Use the StatsPlugin for more control over the generated profile.

`recordsPath`

Description...

?> Add example and description as well as details on `recordsInputPath` and `recordsOutputPath`.

`recordsInputPath`

Description...

?> Add example and description as well as details on `recordsInputPath` and `recordsOutputPath`.

`recordsOutputPath`

Description...

?> Add example and description as well as details on `recordsInputPath` and `recordsOutputPath`.

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

Using `--env`

The webpack CLI support specifying build environment keys via `--cli` such as `--env.production` or `--env.platform=web`. To make use of those settings, change the configuration object into a function in `webpack.config.js`:

```
-module.exports = {
+module.exports = function(env) {
+  return {
    plugins: [
      new webpack.optimize.UglifyJsPlugin({
+        compress: env.production // compress only in production build
      })
    ]
+  };
};
```

Using environment variables

Alternatively, the standard approach in Node.js modules can be applied: Set an environment variable when running webpack and refer to the variables using Node's `process.env`. The variable `NODE_ENV` is commonly used as de-facto standard (see here).

webpack.config.js

```
module.exports = {
  plugins: [
    new webpack.optimize.UglifyJsPlugin({
+      compress: process.env.NODE_ENV === 'production'
    })
  ]
};
```

Use the `cross-env` package to cross-platform-set environment variables:

package.json

```
{
  "scripts": {
    "build": "cross-env NODE_ENV=production PLATFORM=web webpack"
  }
}
```

References

- https://blog.flennik.com/the-fine-art-of-the-webpack-2-config-dc4d19d7f172#.297u8iuz1

webpack has a rich plugin interface. Most of the features within webpack itself use this plugin interface. This makes webpack **flexible**.

Name	Description
`CommonsChunkPlugin`	Generates chunks of common modules shared between entry points and splits them into separate bundles, e.g., `1vendor.bundle.js` && `app.bundle.js`
`ExtractTextWebpackPlugin`	Extracts Text (CSS) from your bundles into a separate file (app.bundle.css)
`ComponentWebpackPlugin`	Use components with webpack
`CompressionWebpackPlugin`	Prepare compressed versions of assets to serve them with Content-Encoding
`I18nWebpackPlugin`	Adds i18n support to your bundles
`HtmlWebpackPlugin`	Simplifies creation of HTML files (`index.html`) to serve your bundles
`NormalModuleReplacementPlugin`	Replaces resource that matches a regexp

For more third-party plugins, see the list from awesome-webpack.

```
new webpack.optimize.CommonsChunkPlugin(options)
```

The `CommonsChunkPlugin` is an opt-in feature that creates a separate file (known as a chunk), consisting of common modules shared between multiple entry points. By separating common modules from bundles, the resulting chunked file can be loaded once initially, and stored in cache for later use. This results in pagespeed optimizations as the browser can quickly serve the shared code from cache, rather than being forced to load a larger bundle whenever a new page is visited.

Options

```
{
  name: string, // or
  names: string[],
  // The chunk name of the commons chunk. An existing chunk can be selected by pas
sing a name of an existing chunk.
  // If an array of strings is passed this is equal to invoking the plugin multipl
e times for each chunk name.
  // If omitted and `options.async` or `options.children` is set all chunks are us
ed,
  // otherwise `options.filename` is used as chunk name.

  filename: string,
  // The filename template for the commons chunk. Can contain the same placeholder
 as `output.filename`.
  // If omitted the original filename is not modified (usually `output.filename` o
r `output.chunkFilename`).

  minChunks: number|Infinity|function(module, count) -> boolean,
  // The minimum number of chunks which need to contain a module before it's moved
 into the commons chunk.
  // The number must be greater than or equal 2 and lower than or equal to the num
ber of chunks.
  // Passing `Infinity` just creates the commons chunk, but moves no modules into
it.
  // By providing a `function` you can add custom logic. (Defaults to the number o
f chunks)

  chunks: string[],
  // Select the source chunks by chunk names. The chunk must be a child of the com
mons chunk.
  // If omitted all entry chunks are selected.

  children: boolean,
  // If `true` all children of the commons chunk are selected
```

```
  async: boolean|string,
  // If `true` a new async commons chunk is created as child of `options.name` and
 sibling of `options.chunks`.
  // It is loaded in parallel with `options.chunks`. It is possible to change the
name of the output file
  // by providing the desired string instead of `true`.

  minSize: number,
  // Minimum size of all common module before a commons chunk is created.
}
```

T> The deprecated webpack 1 constructor `new webpack.optimize.CommonsChunkPlugin(options, filenameTemplate, selectedChunks, minChunks)` is no longer supported. Use a corresponding options object instead.

Examples

Commons chunk for entries

Generate an extra chunk, which contains common modules shared between entry points.

```
new webpack.optimize.CommonsChunkPlugin({
  name: "commons",
  // (the commons chunk name)

  filename: "commons.js",
  // (the filename of the commons chunk)

  // minChunks: 3,
  // (Modules must be shared between 3 entries)

  // chunks: ["pageA", "pageB"],
  // (Only use these entries)
})
```

You must load the generated chunk before the entry point:

```
<script src="commons.js" charset="utf-8"></script>
<script src="entry.bundle.js" charset="utf-8"></script>
```

Explicit vendor chunk

Split your code into vendor and application.

```
entry: {
  vendor: ["jquery", "other-lib"],
  app: "./entry"
}
new webpack.optimize.CommonsChunkPlugin({
  name: "vendor",

  // filename: "vendor.js"
  // (Give the chunk a different name)

  minChunks: Infinity,
  // (with more entries, this ensures that no other module
  //  goes into the vendor chunk)
})
```

```
<script src="vendor.js" charset="utf-8"></script>
<script src="app.js" charset="utf-8"></script>
```

Hint: In combination with long term caching you may need to use the `ChunkManifestWebpackPlugin` to avoid that the vendor chunk changes. You should also use records to ensure stable module ids.

Move common modules into the parent chunk

With Code Splitting multiple child chunks of a chunk can have common modules. You can move these common modules into the parent (This reduces overall size, but has a negative effect on the initial load time. It can be useful if it is expected that a user need to download many sibling chunks).

```
new webpack.optimize.CommonsChunkPlugin({
  // names: ["app", "subPageA"]
  // (choose the chunks, or omit for all chunks)

  children: true,
  // (select all children of chosen chunks)

  // minChunks: 3,
  // (3 children must share the module before it's moved)
})
```

Extra async commons chunk

Similar to the above one, but instead of moving common modules into the parent (which increases initial load time) a new async-loaded additional commons chunk is used. This is automatically downloaded in parallel when the additional chunk is downloaded.

```
new webpack.optimize.CommonsChunkPlugin({
  // names: ["app", "subPageA"]
  // (choose the chunks, or omit for all chunks)

  children: true,
  // (use all children of the chunk)

  async: true,
  // (create an async commons chunk)

  // minChunks: 3,
  // (3 children must share the module before it's separated)
})
```

Passing the `minChunks` property a function

You also have the ability to pass the `minChunks` property a function. This function is called by the `CommonsChunkPlugin` and calls the function with `module` and `count` arguments.

The `module` property represents each module in the chunks you have provided via the `names` property.

The `count` property represents how many chunks the `module` is used in.

This option is useful when you want to have fine-grained control over how the CommonsChunk algorithm determins where modules should be moved to.

```
new webpack.optimize.CommonsChunkPlugin({
  name: "my-single-lib-chunk",
  filename: "my-single-lib-chunk.js",
  minChunks: function(module, countOfHowManyTimesThisModuleIsUsedAcrossAllChunks)
{
    // If module has a path, and inside of the path exists the name "somelib",
    // and it is used in 3 separate chunks/entries, then break it out into
    // a separate chunk with chunk keyname "my-single-lib-chunk", and filename "my
-single-lib-chunk.js"
    return module.resource && (/somelib/).test(module.resource) && count === 3;
  }
});
```

As seen above, this example allows you to move only one lib to a separate file if and only if all conditions are met inside the function.

```
new webpack.DefinePlugin(definitions)
```

The `DefinePlugin` allows you to create global constants which can be configured at **compile** time. This can be useful for allowing different behaviour between development builds and release builds. For example, you might use a global constant to determine whether logging takes place; perhaps you perform logging in your development build but not in the release build. That's the sort of scenario the `DefinePlugin` facilitates.

Example:

```
new webpack.DefinePlugin({
  PRODUCTION: JSON.stringify(true),
  VERSION: JSON.stringify("5fa3b9"),
  BROWSER_SUPPORTS_HTML5: true,
  TWO: "1+1",
  "typeof window": JSON.stringify("object")
})
```

```
console.log("Running App version " + VERSION);
if(!BROWSER_SUPPORTS_HTML5) require("html5shiv");
```

T> Note that because the plugin does a direct text replacement, the value given to it must include actual quotes inside of the string itself. Typically, this is done either with alternate quotes, such as `'"production"'`, or by using `JSON.stringify('production')`.

Each key passed into `DefinePlugin` is an identifier or multiple identifiers joined with `.`.

- If the value is a string it will be used as a code fragment.
- If the value isn't a string, it will be stringified (including functions).
- If the value is an object all keys are defined the same way.
- If you prefix `typeof` to the key, it's only defined for typeof calls.

The values will be inlined into the code which allows a minification pass to remove the redundant conditional.

Example:

```
if (!PRODUCTION) {
  console.log('Debug info')
}
if (PRODUCTION) {
  console.log('Production log')
}
```

```
``
```

After passing through webpack with no minification results in:

```
if (!true) {
  console.log('Debug info')
}
if (true) {
  console.log('Production log')
}
```

and then after a minification pass results in:

```
console.log('Production log')
```

Use Case: Feature Flags

Enable/disable features in production/development build using feature flags.

```
new webpack.DefinePlugin({
  'NICE_FEATURE': JSON.stringify(true),
  'EXPERIMENTAL_FEATURE': JSON.stringify(false)
})
```

Use Case: Service URLs

Use a different service URL in production/development builds:

```
new webpack.DefinePlugin({
  'SERVICE_URL': JSON.stringify("http://dev.example.com")
})
```

The `EnvironmentPlugin` is a shorthand for using the `DefinePlugin` on `process.env` keys.

Usage

The `EnvironmentPlugin` accepts either an array of keys.

```
new webpack.EnvironmentPlugin(['NODE_ENV', 'DEBUG'])
```

This is equivalent to the following `DefinePlugin` application:

```
new webpack.DefinePlugin({
  'process.env.NODE_ENV': JSON.stringify(process.env.NODE_ENV),
  'process.env.DEBUG': JSON.stringify(process.env.DEBUG)
})
```

T> Not specifying the environment variable raises an "`EnvironmentPlugin` - `${key}` environment variable is undefined" error.

Usage with default values

Alternatively, the `EnvironmentPlugin` supports an object, which maps keys to their default values. The default value for a key is taken if the key is undefined in `process.env`.

```
new webpack.EnvironmentPlugin({
  NODE_ENV: 'development', // use 'development' unless process.env.NODE_ENV is def
ined
  DEBUG: false
})
```

W> Variables coming from `process.env` are always strings.

T> Unlike `DefinePlugin`, default values are applied to `JSON.stringify` by the `EnvironmentPlugin`.

T> To specify an unset default value, use `null` instead of `undefined`.

Example:

Let's investigate the result when running the previous `EnvironmentPlugin` configuration on a test file `entry.js`:

```
if (process.env.NODE_ENV === 'production') {
  console.log('Welcome to production');
}
if (process.env.DEBUG) {
  console.log('Debugging output');
}
```

When executing `NODE_ENV=production webpack` in the terminal to build, `entry.js` becomes this:

```
if ('production' === 'production') { // <-- 'production' from NODE_ENV is taken
  console.log('Welcome to production');
}
if (false) { // <-- default value is taken
  console.log('Debugging output');
}
```

Running `DEBUG=false webpack` yields:

```
if ('development' === 'production') { // <-- default value is taken
  console.log('Welcome to production');
}
if ('false') { // <-- 'false' from DEBUG is taken
  console.log('Debugging output');
}
```

The `HtmlWebpackPlugin` simplifies creation of HTML files to serve your webpack bundles. This is especially useful for webpack bundles that include a hash in the filename which changes every compilation. You can either let the plugin generate an HTML file for you, supply your own template using lodash templates, or use your own loader.

Installation

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

Basic Usage

The plugin will generate an HTML5 file for you that includes all your webpack bundles in the body using `script` tags. Just add the plugin to your webpack config as follows:

```
var HtmlWebpackPlugin = require('html-webpack-plugin');
var webpackConfig = {
  entry: 'index.js',
  output: {
    path: 'dist',
    filename: 'index_bundle.js'
  },
  plugins: [new HtmlWebpackPlugin()]
};
```

This will generate a file `dist/index.html` containing the following:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>webpack App</title>
  </head>
  <body>
    <script src="index_bundle.js"></script>
  </body>
</html>
```

If you have multiple webpack entry points, they will all be included with `script` tags in the generated HTML.

If you have any CSS assets in webpack's output (for example, CSS extracted with the ExtractTextPlugin) then these will be included with `<link>` tags in the HTML head.

Configuration

For all configuration options, please see the plugin documentation.

Third party addons

The plugin supports addons. For a list see the documentation.

?> Review this content

The `LoaderOptionsPlugin` is unlike other plugins. It exists to help people move from webpack 1 to webpack 2. With webpack 2 the schema for a `webpack.config.js` became stricter; no longer open for extension by other loaders / plugins. With webpack 2 the intention is that you pass `options` directly to loaders / plugins. i.e. options are **not** global / shared.

However, until a loader has been updated to depend upon options being passed directly to them, the `LoaderOptionsPlugin` exists to bridge the gap. You can configure global / shared loader options with this plugin and all loaders will receive these options.

In the future this plugin may be removed.

```
new webpack.LoaderOptionsPlugin(options)
```

- `options.debug` (`boolean`): Whether loaders should be in `debug` mode or not. `debug` will be removed as of webpack 3.
- `options.minimize` (`boolean`): Where loaders can be switched to minimize mode.
- `options.options` (`object`): A configuration object that can be used to configure older loaders - this will take the same schema a `webpack.config.js`
- `options.options.context` (`string`): The context that can be used to configure older loaders
- other options as in a `webpack.config.js` ....

Examples

```
new webpack.LoaderOptionsPlugin({
  minimize: true,
  debug: false,
  options: {
    context: __dirname
  }
})
```

```
new webpack.ProvidePlugin({identifier1: 'module1', /* ... */})
```

Automatically loads modules. Whenever the `identifier` is encountered as free variable in a module, the `module` is loaded automatically and the `identifier` is filled with the exports of the loaded `module`.

Typical use-cases

Use jQuery

```
new webpack.ProvidePlugin({
  $: 'jquery',
  jQuery: 'jquery'
})
```

```
// in a module
$('#item'); // <= just works
jQuery('#item'); // <= just works
// $ is automatically set to the exports of module "jquery"
```

Use jQuery with Angular 1

Angular looks for `window.jQuery` in order to determine whether jQuery is present, see the source code

```
new webpack.ProvidePlugin({
  'window.jQuery': 'jquery'
})
```

?> Review this content

Adds SourceMaps for assets.

```
new webpack.SourceMapDevToolPlugin(options)
```

- `options.test` / `options.include` / `options.exclude` (`string|RegExp|Array`): Used to determine which assets should be processed. Each one can be a `RegExp` (asset filename is matched), a `string` (asset filename need to start with this string) or an `Array` of those (any of them need to be matched). `test` defaults to `.js` files if omitted.
- `options.filename` (`string`): defines the output filename of the SourceMap. If no value is provided the SourceMap is inlined.
- `options.append` (`string`): is appended to the original asset. Usually the `#sourceMappingURL` comment. `[url]` is replaced with a URL to the SourceMap file. `false` disables the appending.
- `options.moduleFilenameTemplate` / `options.fallbackModuleFilenameTemplate` (`string`): see `output.devtoolModuleFilenameTemplate` .
- `options.module` (`boolean`): (defaults to `true`) When `false` loaders do not generate SourceMaps and the transformed code is used as source instead.
- `options.columns` (`boolean`): (defaults to `true`) When `false` column mappings in SourceMaps are ignored and a faster SourceMap implementation is used.
- `options.lineToLine` (`{test: string|RegExp|Array, include: string|RegExp|Array, exclude: string|RegExp|Array}` matched modules uses simple (faster) line to line source mappings.

Examples

?> TODO

> TODO

webpack provides a Command Line Interface (CLI) to configure and interact with your build. This is mostly useful in case of early prototyping, profiling, writing npm scripts or personal customization of the build.

For proper usage and easy distribution of this configuration, webpack can be configured with `webpack.config.js`. Any parameters sent to the CLI will map to a corresponding parameter in the config file.

Installation

Have a look at this page

?> The new CLI for webpack is under development. New features are being added such as the `--init` flag. Check it out!

Common Usage

```
webpack <entry> [<entry>] <output>
```

Parameter	Configuration Mapping	Explanation
entry	entry	A filename or a set of named filenames which act as the entry point to build your project. If you pass a pair in the form of `=` you can create an additional entry point.
output	output.path + output.filename	A path and filename for the bundled file to be saved in.

Examples

If your project structure is as follows -

```
.
├── dist
├── index.html
└── src
    ├── index.js
    ├── index2.js
    └── others.js
```

```
webpack src/index.js dist/bundle.js
    This will bundle your source code with entry as `index.js` and the output bund
le file will have a path of `dist` and the filename will be `bundle.js`

    | Asset     | Size    | Chunks      | Chunk Names |
    |-----------|---------|-------------|-------------|
    | bundle.js | 1.54 kB | 0 [emitted] | index       |
    [0] ./src/index.js 51 bytes {0} [built]
    [1] ./src/others.js 29 bytes {0} [built]
```

```
webpack index=./src/index.js entry2=./src/index2.js dist/bundle.js
    This will form the bundle with both the files as separate entry points.

    | Asset     | Size    | Chunks        | Chunk Names   |
    |-----------|---------|---------------|---------------|
    | bundle.js | 1.55 kB | 0,1 [emitted] | index, entry2 |
    [0] ./src/index.js 51 bytes {0} [built]
    [0] ./src/index2.js 54 bytes {1} [built]
    [1] ./src/others.js 29 bytes {0} {1} [built]
```

Common Options

List all of the options available on the cli

```
webpack --help , webpack -h
```

Build source using a config file

Specifies a different configuration file to pick up. Use this if you want to specify something different than `webpack.config.js`, which is the default.

```
webpack --config example.config.js
```

Send environment variable to be used in webpack config file

```
webpack --env=DEVELOPMENT
```

Print result of webpack as a JSON

In every other case, webpack prints out a set of stats showing bundle, chunk and timing details. Using this option the output can be a JSON object.This response is accepted by webpack's analyse tool. The analyse tool will take in the JSON and provide all the details

of the build in graphical form.

?> (TODO: Link to webpack analyse article)

```
webpack --json , webpack -j, webpack -j > stats.json
```

Output Options

This set of options allows you to manipulate certain output parameters of your build.

Parameter	Explanation	Input type	Default value
--output-chunk-filename	The output filename for additional chunks	string	filename with [id] instead of [name] or [id] prefixed
--output-filename	The output filename of the bundle	string	[name].js
--output-jsonp-function	The name of the JSONP function used for chunk loading	string	webpackJsonp
--output-library	Expose the exports of the entry point as library	string	
--output-library-target	The type for exposing the exports of the entry,point as library	string	var
--output-path	The output path for compilation assets	string	Current directory
--output-pathinfo	Include a comment with the request for every dependency	boolean	false
--output-public-path	The public path for the assets	string	/
--output-source-map-filename	The output filename for the SourceMap	string	[name].map or [outputFilename].map

Example Usage

```
webpack index=./src/index.js index2=./src/index2.js --output-path='./dist' --outpu
t-filename='[name][hash].bundle.js'

| Asset                                 | Size    | Chunks      | Chunk Names   |
|---------------------------------------|---------|-------------|---------------|
| index2740fdca26e9348bedbec.bundle.js |  2.6 kB | 0 [emitted] | index2        |
| index740fdca26e9348bedbec.bundle.js  | 2.59 kB | 1 [emitted] | index         |
    [0] ./src/others.js 29 bytes {0} {1} [built]
    [1] ./src/index.js 51 bytes {1} [built]
    [2] ./src/index2.js 54 bytes {0} [built]
```

```
webpack.js index=./src/index.js index2=./src/index2.js --output-path='./dist' --ou
tput-filename='[name][hash].bundle.js' --devtool source-map --output-source-map-fi
lename='[name]123.map'

| Asset                                 | Size    | Chunks      | Chunk Names   |
|---------------------------------------|---------|-------------|---------------|
| index2740fdca26e9348bedbec.bundle.js | 2.76 kB | 0 [emitted] | index2        |
|  index740fdca26e9348bedbec.bundle.js | 2.74 kB | 1 [emitted] | index         |
|                        index2123.map | 2.95 kB | 0 [emitted] | index2        |
|                         index123.map | 2.95 kB | 1 [emitted] | index         |
    [0] ./src/others.js 29 bytes {0} {1} [built]
    [1] ./src/index.js 51 bytes {1} [built]
    [2] ./src/index2.js 54 bytes {0} [built]
```

Debug Options

This set of options allows you to better debug the application containing assets compiled with webpack

Parameter	Explanation	Input type	Default value
--debug	Switch loaders to debug mode	boolean	false
--devtool	Define source map type for the bundled resources	string	-
--progress	Print compilation progress in percentage	boolean	false

Module Options

These options allow you to bind modules as allowed by webpack

Parameter	Explanation	Usage

--module-bind	Bind an extension to a loader	--module-bind /.js$/=babel
--module-bind-post	Bind an extension to a post loader	
--module-bind-pre	Bind an extension to a pre loader	

Watch Options

These options makes the build watch for changes in files of the dependency graph and perform the build again.

Parameter	Explanation
--watch, -w	Watch the filesystem for changes
--save, -s	Recompiles on save regardless of changes
--watch-aggregate-timeout	Timeout for gathering changes while watching
--watch-poll	The polling interval for watching (also enable polling)
--watch-stdin, --stdin	Exit the process when stdin is closed

Optimize Options

These options allow you to manipulate optimisations for a production build using webpack

Parameter	Explanation	Plugin used
--optimize-max-chunks	Try to keep the chunk count below a limit	LimitChunkCountPlugin
--optimize-min-chunk-size	Try to keep the chunk size above a limit	MinChunkSizePlugin
--optimize-minimize	Minimize javascript and switches loaders to minimizing	UglifyJsPlugin & LoaderOptionsPlugin

Resolve Options

These allow you to configure the webpack resolver with aliases and extensions.

Parameter	Explanation	Example
--resolve-alias	Setup a module alias for resolving	--resolve-alias jquery-plugin=jquery.plugin
--resolve-extensions	Setup extensions that should be used to resolve,modules	--resolve-extensions .es6 .js .ts
--resolve-loader-alias	Minimize javascript and switches loaders to minimizing	

Stats Options

These options allow webpack to display various stats and style them differently in the console output.

Parameter	Explanation	Type
--color, --colors	Enables/Disables colors on the console [default: (supports-color)]	boolean
--display-cached	Display also cached modules in the output	boolean
--display-cached-assets	Display also cached assets in the output	boolean
--display-chunks	Display chunks in the output	boolean
--display-entrypoints	Display entry points in the output	boolean
--display-error-details	Display details about errors	boolean
--display-exclude	Exclude modules in the output	boolean
--display-modules	Display even excluded modules in the output	boolean
--display-origins	Display origins of chunks in the output	boolean
--display-reasons	Display reasons about module inclusion in the output	boolean
--display-used-exports	Display information about used exports in modules (Tree Shaking)	boolean

--hide-modules	Hides info about modules	boolean
--sort-assets-by	Sorts the assets list by property in asset	string
--sort-chunks-by	Sorts the chunks list by property in chunk	string
--sort-modules-by	Sorts the modules list by property in module	string
--verbose, -v	Show more details	boolean

Advanced Options

Parameter	Explanation	Usage
--bail	Abort the compilation on first error	
--cache	Enable in memory caching [Enabled by default for watch]	--cache=false
--define	Define any free var in the bundle	--define process.env.NODE_ENV='develo
--hot	Enables Hot Module Replacement [Uses HotModuleReplacementPlugin]	--hot=true
--labeled-modules	Enables labeled modules [Uses LabeledModulesPlugin]	
--plugin	Load this plugin	
--prefetch	Prefetch the particular file	--prefetch=./files.js
--provide	Provide these modules as free vars in all modules	--provide jQuery=jquery
--records-input-path	Path to the records file (reading)	
--records-output-path	Path to the records file (writing)	
--records-path	Path to the records file	
--target	The targeted execution environment	--target='node'

Shortcuts

Shortcut	Replaces
-d	--debug --devtool eval-cheap-module-source-map --output-pathinfo
-p	--optimize-minimize --define process.env.NODE_ENV="production"

Profiling

This option profiles the compilation and includes this information in the stats output. It gives you an in depth idea of which step in the compilation is taking how long. This can help you optimise your build in a more informed manner.

```
webpack --profile

30ms building modules
1ms sealing
1ms optimizing
0ms basic module optimization
1ms module optimization
1ms advanced module optimization
0ms basic chunk optimization
0ms chunk optimization
1ms advanced chunk optimization
0ms module and chunk tree optimization
1ms module reviving
0ms module order optimization
1ms module id optimization
1ms chunk reviving
0ms chunk order optimization
1ms chunk id optimization
10ms hashing
0ms module assets processing
13ms chunk assets processing
1ms additional chunk assets processing
0ms recording
0ms additional asset processing
26ms chunk asset optimization
1ms asset optimization
6ms emitting
```

Loaders allow you to preprocess files as you `require()` or “load” them. Loaders are kind of like “tasks” in other build tools, and provide a powerful way to handle front-end build steps. Loaders can transform files from a different language (like CoffeeScript to JavaScript), or inline images as data URLs. Loaders even allow you to do things like `require()` css files right in your JavaScript!

To tell webpack to transform a module with a loader, you can specify the loader in the webpack configuration file (preferred) or in the module request, such as in a `require()` call.

?> When /concepts/loaders merges, we should link to the many usages of loaders found there (require vs configuration) from this page.

How to write a loader

A loader is just a JavaScript module that exports a function. The compiler calls this function and passes the result of the previous loader or the resource file into it. The `this` context of the function is filled-in by the compiler with some useful methods that allow the loader (among other things) to change its invocation style to async, or get query parameters. The first loader is passed one argument: the content of the resource file. The compiler expects a result from the last loader. The result should be a `String` or a `Buffer` (which is converted to a string), representing the JavaScript source code of the module. An optional SourceMap result (as JSON object) may also be passed.

A single result can be returned in **sync mode**. For multiple results the `this.callback()` must be called. In **async mode** `this.async()` must be called. It returns `this.callback()` if **async mode** is allowed. Then the loader must return `undefined` and call the callback.

Examples

Sync Loader

sync-loader.js

```
module.exports = function(content) {
    return someSyncOperation(content);
};
```

Async Loader

async-loader.js

```
module.exports = function(content) {
    var callback = this.async();
    if(!callback) return someSyncOperation(content);
    someAsyncOperation(content, function(err, result) {
        if(err) return callback(err);
        callback(null, result);
    });
};
```

T> It's recommended to give an asynchronous loader a fall-back to synchronous mode. This isn't required for webpack, but allows the loader to run synchronously using enhanced-require.

"Raw" Loader

By default, the resource file is treated as utf-8 string and passed as String to the loader. By setting raw to true the loader is passed the raw `Buffer`. Every loader is allowed to deliver its result as `String` or as `Buffer`. The compiler converts them between loaders.

raw-loader.js

```
module.exports = function(content) {
    assert(content instanceof Buffer);
    return someSyncOperation(content);
    // return value can be a `Buffer` too
    // This is also allowed if loader is not "raw"
};
module.exports.raw = true;
```

Pitching Loader

The order of chained loaders are **always** called from right to left. But, in some cases, loaders do not care about the results of the previous loader or the resource. They only care for **metadata**. The `pitch` method on the loaders is called from **left to right** before the loaders are called (from right to left).

If a loader delivers a result in the `pitch` method the process turns around and skips the remaining loaders, continuing with the calls to the more left loaders. `data` can be passed between pitch and normal call.

```
module.exports = function(content) {
    return someSyncOperation(content, this.data.value);
};
module.exports.pitch = function(remainingRequest, precedingRequest, data) {
    if(someCondition()) {
        // fast exit
        return "module.exports = require(" + JSON.stringify("-!" + remainingReques
t) + ");";
    }
    data.value = 42;
};
```

The loader context

The loader context represents the properties that are available inside of a loader assigned to the `this` property.

Given the following example this require call is used: In `/abc/file.js`:

```
require("./loader1?xyz!loader2!./resource?rrr");
```

`version`

Loader API version. Currently `2`. This is useful for providing backwards compatibility. Using the version you can specify custom logic or fallbacks for breaking changes.

`context`

The directory of the module. Can be used as context for resolving other stuff.

In the example: `/abc` because `resource.js` is in this directory

`request`

The resolved request string.

In the example: `"/abc/loader1.js?xyz!/abc/node_modules/loader2/index.js!/abc/resource.js?rrr"`

query

A string. The query of the request for the current loader.

In the example: in loader1: `"?xyz"`, in loader2: `""`

data

A data object shared between the pitch and the normal phase.

cacheable

```
cacheable(flag = true: boolean)
```

By default, loader results are cacheable. Call this method passing `false` to make the loader's result not cacheable.

A cacheable loader must have a deterministic result, when inputs and dependencies haven't changed. This means the loader shouldn't have other dependencies than specified with `this.addDependency`. Most loaders are deterministic and cacheable.

loaders

```
loaders = [{request: string, path: string, query: string, module: function}]
```

An array of all the loaders. It is writeable in the pitch phase.

In the example:

```
[
  { request: "/abc/loader1.js?xyz",
    path: "/abc/loader1.js",
    query: "?xyz",
    module: [Function]
  },
  { request: "/abc/node_modules/loader2/index.js",
    path: "/abc/node_modules/loader2/index.js",
    query: "",
    module: [Function]
```

```
  }
]
```

`loaderIndex`

The index in the loaders array of the current loader.

In the example: in loader1: `0`, in loader2: `1`

`resource`

The resource part of the request, including query.

In the example: `"/abc/resource.js?rrr"`

`resourcePath`

The resource file.

In the example: `"/abc/resource.js"`

`resourceQuery`

The query of the resource.

In the example: `"?rrr"`

`emitWarning`

```
emitWarning(message: string)
```

Emit a warning.

`emitError`

```
emitError(message: string)
```

Emit an error.

exec

```
exec(code: string, filename: string)
```

Execute some code fragment like a module.

T> Don't use `require(this.resourcePath)`, use this function to make loaders chainable!

resolve

```
resolve(context: string, request: string, callback: function(err, result: string))
```

Resolve a request like a require expression.

resolveSync

```
resolveSync(context: string, request: string) -> string
```

Resolve a request like a require expression.

addDependency

```
addDependency(file: string)
dependency(file: string) // shortcut
```

Adds a file as dependency of the loader result in order to make them watchable. For example, `html-loader` uses this technique as it finds `src` and `src-set` attributes. Then, it sets the url's for those attributes as dependencies of the html file that is parsed.

addContextDependency

```
addContextDependency(directory: string)
```

Add a directory as dependency of the loader result.

clearDependencies

```
clearDependencies()
```

Remove all dependencies of the loader result. Even initial dependencies and these of other loaders. Consider using `pitch`.

`value`

Pass values to the next loader. If you know what your result exports if executed as module, set this value here (as a only element array).

`inputValue`

Passed from the last loader. If you would execute the input argument as module, consider reading this variable for a shortcut (for performance).

`options`

The options passed to the Compiler.

`debug`

A boolean flag. It is set when in debug mode.

`minimize`

Should the result be minimized.

`sourceMap`

Should a SourceMap be generated.

`target`

Target of compilation. Passed from configuration options.

Example values: `"web"`, `"node"`

`webpack`

This boolean is set to true when this is compiled by webpack.

T> Loaders were originally designed to also work as Babel transforms. Therefore if you write a loader that works for both, you can use this property to know if there is access to additional loaderContext and webpack features.

`emitFile`

```
emitFile(name: string, content: Buffer|String, sourceMap: {...})
```

Emit a file. This is webpack-specific.

`fs`

Access to the `compilation`'s `inputFileSystem` property.

`_compilation`

Hacky access to the Compilation object of webpack.

`_compiler`

Hacky access to the Compiler object of webpack.

`_module`

Hacky access to the Module object being loaded.

Custom `loaderContext` Properties

Custom properties can be added to the `loaderContext` by either specifying values on the `loader` proprty on your webpack configuration, or by creating a custom plugin that hooks into the `normal-module-loader` event which gives you access to the `loaderContext` to modify or extend.

webpack provides a Node.js API which can be used directly in Node.js runtime.

The Node.js API is useful in scenarios in which you need to customize the build or development process since all the reporting and error handling must be done manually and webpack only does the compiling part. For this reason the `stats` configuration options will not have any effect in the `webpack()` call.

Installation

To start using webpack Node.js API, first install webpack if you haven't yet:

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

Then require the webpack module in your Node.js script:

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

// Or if you prefer ES2015:
import webpack from "webpack";
```

`webpack()`

The imported `webpack` function is fed a webpack Configuration Object and runs the webpack compiler if a callback function is provided:

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

webpack({
  // [Configuration Object](/configuration/)
}, (err, [stats](#stats-object)) => {
  if (err || stats.hasErrors()) {
    // [Handle errors here](#error-handling)
  }
  // Done processing
});
```

T> The `err` object **will not** include compilation errors and those must be handled separately using `stats.hasErrors()` which will be covered in detail in Error Handling section of this guide. The `err` object will only contain webpack-related issues, such as misconfiguration, etc.

Note that you can provide the `webpack` function with an array of configurations:

```
webpack([
  { /* Configuration Object */ },
  { /* Configuration Object */ },
  { /* Configuration Object */ }
], (err, [stats](#stats-object)) => {
  // ...
});
```

T> webpack will **not** run the multiple configurations in parallel. Each configuration is only processed after the previous one has finished processing. To have webpack process them in parallel, you can use a third-party solution like parallel-webpack.

Compiler Instance

If you don't pass the `webpack` runner function a callback, it will return a webpack `Compiler` instance. This instance can be used to manually trigger the webpack runner or have it build and watch for changes. Much like the CLI Api. The `Compiler` instance provides the following methods:

- `.run(callback)`
- `.watch(watchOptions, handler)`

Run

Calling the `run` method on the `Compiler` instance is much like the quick run method mentioned above:

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

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

compiler.run((err, [stats](#stats-object)) => {
  // ...
});
```

Watching

Calling the `watch` method, triggers the webpack runner, but then watches for changes (much like CLI: `webpack --watch`), as soon as webpack detects a change, runs again. Returns an instance of `Watching`.

```
watch(watchOptions, callback)
```

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

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

const watching = compiler.watch({
  <details><summary>/* [watchOptions](/configuration/watch/#watchoptions) */</summ
ary>
  aggregateTimeout: 300,
  poll: undefined
  </details>
}, (err, [stats](#stats-object)) => {
  // Print watch/build result here...
  console.log(stats);
});
```

`Watching` options are covered in detail here.

Close `Watching`

The `watch` method returns a `Watching` instance that exposes `.close(callback)` method. Calling this method will end watching:

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

T> It's not allowed to watch or run again before the existing watcher has been closed or invalidated.

Invalidate `Watching`

Manually invalidate the current compiling round, but don't stop watching.

```
watching.invalidate(() => {
```

```
  console.warn("Invalidated.");
});
```

Stats Object

The `stats` object that is passed as a second argument of the `webpack()` callback, is a good source of information about the code compilation process. It includes:

- Errors and Warnings (if any)
- Timings
- Module and Chunk information
- etc.

The webpack CLI uses this information to display a nicely formatted output in your console.

This object exposes these methods:

`stats.hasErrors()`

Can be used to check if there were errors while compiling. Returns `true` or `false`.

`stats.hasWarnings()`

Can be used to check if there were warnings while compiling. Returns `true` or `false`.

`stats.toJson(options)`

Returns compilation information as a JSON object. `options` can be either a string (a preset) or an object for more granular control:

```
stats.toJson("minimal"); // [more options: "verbose", etc](/configuration/stats).
```

```
stats.toJson({
  assets: false,
  hash: true
});
```

All available options and presets are described in Stats documentation

> Here's an example of this function's output

`stats.toString(options)`

Returns a formatted string of the compilation information (similar to CLI output).

Options are the same as `stats.toJson(options)` with one addition:

```
stats.toString({
  // ...
  // Add console colors
  colors: true
});
```

Here's an example of `stats.toString()` usage:

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

webpack({
  // [Configuration Object](/configuration/)
}, (err, stats) => {
  if (err) {
    console.error(err);
    return;
  }

  console.log(stats.toString({
    chunks: false,  // Makes the build much quieter
    colors: true    // Shows colors in the console
  }));
});
```

Error Handling

For a good error handling, you need to account for these three types of errors:

- Fatal webpack errors (wrong configuration, etc)
- Compilation errors (missing modules, syntax errors, etc)
- Compilation warnings

Here's an example that does all that:

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

```
webpack({
  // [Configuration Object](/configuration/)
}, (err, stats) => {
  if (err) {
    console.error(err.stack || err);
    if (err.details) {
      console.error(err.details);
    }
    return;
  }

  const info = stats.toJson();

  if (stats.hasErrors()) {
    console.error(info.errors);
  }

  if (stats.hasWarnings()) {
    console.warn(info.warnings)
  }

  // Log result...
});
```

Compiling to Memory

webpack writes the output to the specified files on disk. If you want webpack to output them to a different kind of file system (memory, webDAV, etc), you can set the `outputFileSystem` option on the compiler:

```
const MemoryFS = require("memory-fs");
const webpack = require("webpack");

const fs = new MemoryFS();
const compiler = webpack({ /* options*/ });

compiler.outputFileSystem = fs;
compiler.run((err, stats) => {
  // Read the output later:
  const content = fs.readFileSync("...");
});
```

T> The output file system you provide needs to be compatible with Node's own `fs` module interface.

For a high-level introduction to writing plugins, start with How to write a plugin.

Many objects in webpack extend the Tapable class, which exposes a `plugin` method. And with the `plugin` method, plugins can inject custom build steps. You will see `compiler.plugin` and `compilation.plugin` used a lot. Essentially, each one of these plugin calls binds a callback to fire at specific steps throughout the build process.

A plugin is installed once as webpack starts up. webpack installs a plugin by calling its `apply` method, and passes a reference to the webpack `compiler` object. You may then call `compiler.plugin` to access asset compilations and their individual build steps. An example would look like this:

```
// MyPlugin.js

function MyPlugin(options) {
  // Configure your plugin with options...
}

MyPlugin.prototype.apply = function(compiler) {
  compiler.plugin("compile", function(params) {
    console.log("The compiler is starting to compile...");
  });

  compiler.plugin("compilation", function(compilation) {
    console.log("The compiler is starting a new compilation...");

    compilation.plugin("optimize", function() {
      console.log("The compilation is starting to optimize files...");
    });
  });

  compiler.plugin("emit", function(compilation, callback) {
    console.log("The compilation is going to emit files...");
    callback();
  });
};

module.exports = MyPlugin;
```

Then in `webpack.config.js`

```
    plugins: [
        new MyPlugin({options: 'nada'})
    ]
```

Plugin Interfaces

There are two types of plugin interfaces.

- Timing based
  - sync (default): As seen above. Use return.
  - async: Last parameter is a callback. Signature: function(err, result)
  - parallel: The handlers are invoked parallel (async).
- Return value
  - not bailing (default): No return value.
  - bailing: The handlers are invoked in order until one handler returns something.
  - parallel bailing: The handlers are invoked in parallel (async). The first returned value (by order) is significant.
  - waterfall: Each handler gets the result value of the last handler as an argument.

The `Compiler` instance

Plugins need to have the apply method on their prototype chain (or bound to) in order to have access to the compiler instance.

MyPlugin.js

```
function MyPlugin() {};
MyPlugin.prototype.apply = function (compiler) {
    //now you have access to all the compiler instance methods
}
module.exports = MyPlugin;
```

Something like this should also work

MyFunction.js

```
function apply(options, compiler) {
    //now you have access to the compiler instance
    //and options
}

//this little trick makes it easier to pass and check options to the plugin
module.exports = function(options) {
    if (options instanceof Array) {
        options = {
```

```
            include: options
        };
    }

    if (!Array.isArray(options.include)) {
        options.include = [ options.include ];
    }

    return {
        apply: apply.bind(this, options)
    };
};
```

`run(compiler: Compiler)` async

The `run` method of the Compiler is used to start a compilation. This is not called in watch mode.

`watch-run(watching: Watching)` async

The `watch` method of the Compiler is used to start a watching compilation. This is not called in normal mode.

`compilation(c: Compilation, params: Object)`

A `Compilation` is created. A plugin can use this to obtain a reference to the `Compilation` object. The `params` object contains useful references.

`normal-module-factory(nmf: NormalModuleFactory)`

A `NormalModuleFactory` is created. A plugin can use this to obtain a reference to the `NormalModuleFactory` object.

```
compiler.plugin("normal-module-factory", function(nmf) {
    nmf.plugin("after-resolve", function(data) {
        data.loaders.unshift(path.join(__dirname, "postloader.js"));
    });
});
```

`context-module-factory(cmf: ContextModuleFactory)`

A `ContextModuleFactory` is created. A plugin can use this to obtain a reference to the `ContextModuleFactory` object.

`compile(params)`

The Compiler starts compiling. This is used in normal and watch mode. Plugins can use this point to modify the `params` object (i. e. to decorate the factories).

```
compiler.plugin("compile", function(params) {
    //you are now in the "compile" phase
});
```

`make(c: Compilation)` parallel

Plugins can use this point to add entries to the compilation or prefetch modules. They can do this by calling `addEntry(context, entry, name, callback)` or `prefetch(context, dependency, callback)` on the Compilation.

`after-compile(c: Compilation)` async

The compile process is finished and the modules are sealed. The next step is to emit the generated stuff. Here modules can use the results in some cool ways.

The handlers are not copied to child compilers.

`emit(c: Compilation)` async

The Compiler begins with emitting the generated assets. Here plugins have the last chance to add assets to the `c.assets` array.

`after-emit(c: Compilation)` async

The Compiler has emitted all assets.

`done(stats: Stats)`

All is done.

`failed(err: Error)`

The Compiler is in watch mode and a compilation has failed hard.

`invalid()`

The Compiler is in watch mode and a file change is detected. The compilation will be begin shortly (`options.watchDelay`).

`after-plugins()`

All plugins extracted from the options object are added to the compiler.

`after-resolvers()`

All plugins extracted from the options object are added to the resolvers.

The `Compilation` instance

The Compilation instance extends from the compiler. I.e. compiler.compilation It is the literal compilation of all the objects in the require graph. This object has access to all the modules and their dependencies (most of which are circular references). In the compilation phase, modules are loaded, sealed, optimized, chunked, hashed and restored, etc. This would be the main lifecycle of any operations of the compilation.

```
compiler.plugin("compilation", function(compilation) {
    //the main compilation instance
    //all subsequent methods are derived from compilation.plugin
});
```

`normal-module-loader`

The normal module loader, is the function that actually loads all the modules in the module graph (one-by-one).

```
compilation.plugin('normal-module-loader', function(loaderContext, module) {
    //this is where all the modules are loaded
    //one by one, no dependencies are created yet
});
```

seal

The sealing of the compilation has started.

```
compilation.plugin('seal', function() {
    //you are not accepting any more modules
    //no arguments
});
```

optimize

Optimize the compilation.

```
compilation.plugin('optimize', function() {
    //webpack is begining the optimization phase
    // no arguments
});
```

optimize-tree(chunks, modules) async

Async optimization of the tree.

```
compilation.plugin('optimize-tree', function(chunks, modules) {

});
```

optimize-modules(modules: Module[])

Optimize the modules.

```
compilation.plugin('optimize-modules', function(modules) {
    //handle to the modules array during tree optimization
});
```

after-optimize-modules(modules: Module[])

Optimizing the modules has finished.

optimize-chunks(chunks: Chunk[])

Optimize the chunks.

```
//optimize chunks may be run several times in a compilation

compilation.plugin('optimize-chunks', function(chunks) {
    //unless you specified multiple entries in your config
    //there's only one chunk at this point
    chunks.forEach(function (chunk) {
        //chunks have circular references to their modules
        chunk.modules.forEach(function (module){
            //module.loaders, module.rawRequest, module.dependencies, etc.
        });
    });
});
```

after-optimize-chunks(chunks: Chunk[])

Optimizing the chunks has finished.

revive-modules(modules: Module[], records)

Restore module info from records.

optimize-module-order(modules: Module[])

Sort the modules in order of importance. The first is the most important module. It will get the smallest id.

optimize-module-ids(modules: Module[])

Optimize the module ids.

after-optimize-module-ids(modules: Module[])

Optimizing the module ids has finished.

record-modules(modules: Module[], records)

Store module info to the records.

revive-chunks(chunks: Chunk[], records)

Restore chunk info from records.

`optimize-chunk-order(chunks: Chunk[])`

Sort the chunks in order of importance. The first is the most important chunk. It will get the smallest id.

`optimize-chunk-ids(chunks: Chunk[])`

Optimize the chunk ids.

`after-optimize-chunk-ids(chunks: Chunk[])`

Optimizing the chunk ids has finished.

`record-chunks(chunks: Chunk[], records)`

Store chunk info to the records.

`before-hash`

Before the compilation is hashed.

`after-hash`

After the compilation is hashed.

`before-chunk-assets`

Before creating the chunk assets.

`additional-chunk-assets(chunks: Chunk[])`

Create additional assets for the chunks.

`record(compilation, records)`

Store info about the compilation to the records

`optimize-chunk-assets(chunks: Chunk[])` async

Optimize the assets for the chunks.

The assets are stored in `this.assets`, but not all of them are chunk assets. A `Chunk` has a property `files` which points to all files created by this chunk. The additional chunk assets are stored in `this.additionalChunkAssets`.

Here's an example that simply adds a banner to each chunk.

```
compilation.plugin("optimize-chunk-assets", function(chunks, callback) {
    chunks.forEach(function(chunk) {
        chunk.files.forEach(function(file) {
            compilation.assets[file] = new ConcatSource("\/**Sweet Banner**\/", "\
n", compilation.assets[file]);
        });
    });
    callback();
});
```

`after-optimize-chunk-assets(chunks: Chunk[])`

The chunk assets have been optimized. Here's an example plugin from @boopathi that outputs exactly what went into each chunk.

```
var PrintChunksPlugin = function() {};
PrintChunksPlugin.prototype.apply = function(compiler) {
    compiler.plugin('compilation', function(compilation, params) {
        compilation.plugin('after-optimize-chunk-assets', function(chunks) {
            console.log(chunks.map(function(c) {
                return {
                    id: c.id,
                    name: c.name,
                    includes: c.modules.map(function(m) {
                        return m.request;
                    })
                };
            }));
        });
    });
};
```

`optimize-assets(assets: Object{name: Source})` async

Optimize all assets.

The assets are stored in `this.assets`.

`after-optimize-assets(assets: Object{name: Source})`

The assets has been optimized.

`build-module(module)`

Before a module build has started.

```
compilation.plugin('build-module', function(module){
    console.log('build module');
    console.log(module);
});
```

`succeed-module(module)`

A module has been built successfully.

```
compilation.plugin('succeed-module', function(module){
    console.log('succeed module');
    console.log(module);
});
```

`failed-module(module)`

The module build has failed.

```
compilation.plugin('failed-module', function(module){
    console.log('failed module');
    console.log(module);
});
```

`module-asset(module, filename)`

An asset from a module was added to the compilation.

`chunk-asset(chunk, filename)`

An asset from a chunk was added to the compilation.

The `MainTemplate` instance

`startup(source, module, hash)`

```
    compilation.mainTemplate.plugin('startup', function(source, module, hash) {
      if (!module.chunks.length && source.indexOf('__ReactStyle__') === -1) {
        var originName = module.origins && module.origins.length ? module.origins[0
].name : 'main';
        return ['if (typeof window !== "undefined") {',
            '  window.__ReactStyle__ = ' + JSON.stringify(classNames[originName])
+ ';',
            '}'
          ].join('\n') + source;
      }
      return source;
    });
```

The `Parser` instance (`compiler.parser`)

The parser instance takes a String and callback and will return an expression when there's a match.

```
compiler.plugin('compilation', function(compilation, data) {
  data.normalModuleFactory.plugin('parser', function(parser, options) {
    parser.plugin('call require', function(expr) {
      // you now have a reference to the call expression
    });
  });
});
```

`program(ast)` bailing

General purpose plugin interface for the AST of a code fragment.

`statement(statement: Statement)` bailing

General purpose plugin interface for the statements of the code fragment.

`call <identifier>(expr: Expression)` bailing

`abc(1)` => `call abc`

`a.b.c(1)` => `call a.b.c`

`expression <identifier>(expr: Expression)` bailing

`abc` => `expression abc`

`a.b.c` => `expression a.b.c`

`expression ?:(expr: Expression)` bailing

`(abc ? 1 : 2)` => `expression ?!`

Return a boolean value to omit parsing of the wrong path.

`typeof <identifier>(expr: Expression)` bailing

`typeof a.b.c` => `typeof a.b.c`

`statement if(statement: Statement)` bailing

`if(abc) {}` => `statement if`

Return a boolean value to omit parsing of the wrong path.

`label <labelname>(statement: Statement)` bailing

`xyz: abc` => `label xyz`

`var <name>(statement: Statement)` bailing

`var abc, def` => `var abc` + `var def`

Return `false` to not add the variable to the known definitions.

`evaluate <expression type>(expr: Expression)` bailing

Evaluate an expression.

`evaluate typeof <identifier>(expr: Expression)` bailing

Evaluate the type of an identifier.

`evaluate Identifier <identifier>(expr: Expression)` bailing

Evaluate a identifier that is a free var.

`evaluate defined Identifier <identifier>(expr: Expression)` bailing

Evaluate a identifier that is a defined var.

`evaluate CallExpression .<property>(expr: Expression)` bailing

Evaluate a call to a member function of a successfully evaluated expression.

The `NormalModuleFactory`

`before-resolve(data)` async waterfall

Before the factory starts resolving. The `data` object has these properties:

- `context` The absolute path of the directory for resolving.
- `request` The request of the expression.

Plugins are allowed to modify the object or to pass a new similar object to the callback.

`after-resolve(data)` async waterfall

After the factory has resolved the request. The `data` object has these properties:

- `request` The resolved request. It acts as an identifier for the NormalModule.
- `userRequest` The request the user entered. It's resolved, but does not contain pre or post loaders.
- `rawRequest` The unresolved request.
- `loaders` A array of resolved loaders. This is passed to the NormalModule and they will be executed.
- `resource` The resource. It will be loaded by the NormalModule.
- `parser` The parser that will be used by the NormalModule.

The `ContextModuleFactory`

`before-resolve(data)` async waterfall

`after-resolve(data)` async waterfall

`alternatives(options: Array)` async waterfall

Resolvers

- `compiler.resolvers.normal` Resolver for a normal module
- `compiler.resolvers.context` Resolver for a context module
- `compiler.resolvers.loader` Resolver for a loader

Any plugin should use `this.fileSystem` as fileSystem, as it's cached. It only has async named functions, but they may behave sync, if the user uses a sync file system implementation (i. e. in enhanced-require).

To join paths any plugin should use `this.join`. It normalizes the paths. There is a `this.normalize` too.

A bailing async forEach implementation is available on `this.forEachBail(array, iterator, callback)`.

To pass the request to other resolving plugins, use the `this.doResolve(types: String|String[], request: Request, callback)` or (`this.doResolve(types, request, message, callback)`) method. `types` are multiple possible request types that are tested

in order of preference.

```
interface Request {
  path: String // The current directory of the request
  request: String // The current request string
  query: String // The query string of the request, if any
  module: boolean // The request begins with a module
  directory: boolean // The request points to a directory
  file: boolean // The request points to a file
  resolved: boolean // The request is resolved/done
  // undefined means false for boolean fields
}

// Examples
// from /home/user/project/file.js: require("../test?charset=ascii")
{
  path: "/home/user/project",
  request: "../test",
  query: "?charset=ascii"
}
// from /home/user/project/file.js: require("test/test/")
{
  path: "/home/user/project",
  request: "test/test/",
  module: true,
  directory: true
}
```

`resolve(context: String, request: String)`

Before the resolving process starts.

`resolve-step(types: String[], request: Request)`

Before a single step in the resolving process starts.

`module(request: Request)` async waterfall

A module request is found and should be resolved.

`directory(request: Request)` async waterfall

A directory request is found and should be resolved.

`file(request: Request)` async waterfall

A file request is found and should be resolved.

The plugins may offer more extensions points

Here is a list what the default plugins in webpack offer. They are all `(request: Request)` async waterfall.

The process for normal modules and contexts is `module -> module-module -> directory -> file`.

The process for loaders is `module -> module-loader-module -> module-module -> directory -> file`.

`module-module`

A module should be looked up in a specified directory. `path` contains the directory.

`module-loader-module` (only for loaders)

Used before module templates are applied to the module name. The process continues with `module-module`.

These guides cover what you need to know in order to develop webpack. WIP!

A loader is a node module exporting a `function`.

This function is called when a resource should be transformed by this loader.

In the simple case, when only a single loader is applied to the resource, the loader is called with one parameter: the content of the resource file as string.

The loader can access the loader API on the `this` context in the function.

A sync loader that only wants to give a one value can simply `return` it. In every other case the loader can give back any number of values with the `this.callback(err, values...)` function. Errors are passed to the `this.callback` function or thrown in a sync loader.

The loader is expected to give back one or two values. The first value is a resulting JavaScript code as string or buffer. The second optional value is a SourceMap as JavaScript object.

In the complex case, when multiple loaders are chained, only the last loader gets the resource file and only the first loader is expected to give back one or two values (JavaScript and SourceMap). Values that any other loader give back are passed to the previous loader.

Examples

```
// Identity loader
module.exports = function(source) {
  return source;
};
```

```
// Identity loader with SourceMap support
module.exports = function(source, map) {
  this.callback(null, source, map);
};
```

Guidelines

(Ordered by priority, first one should get the highest priority)

- Loaders should do only a single task

- Loaders can be chained. Create loaders for every step, instead of a loader that does everything at once.

This also means they should not convert to JavaScript if not necessary.

Example: Render HTML from a template file by applying the query parameters

I could write a loader that compiles the template from source, execute it and return a module that exports a string containing the HTML code. This is bad.

Instead I should write loaders for every task in this use case and apply them all (pipeline):

- `jade-loader` : Convert template to a module that exports a function.
- `apply-loader` : Takes a function exporting module and returns raw result by applying query parameters.
- `html-loader` : Takes HTML and exports a string exporting module.

Generate modules that are modular

Loader generated modules should respect the same design principles like normal modules.

Example: That's a bad design: (not modular, global state, ...)

```
require("any-template-language-loader!./xyz.atl");

var html = anyTemplateLanguage.render("xyz");
```

Flag itself cacheable if possible

Most loaders are cacheable, so they should flag itself as cacheable.

Just call `cacheable` in the loader.

```
// Cacheable identity loader
module.exports = function(source) {
  this.cacheable();
  return source;
};
```

Do not keep state between runs and modules

A loader should be independent of other modules compiled (except of these issued by the loader).

A loader should be independent of previous compilations of the same module.

Mark dependencies

If a loader uses external resources (i. e. by reading from filesystem), they **must** tell about that. This information is used to invalidate cacheable loaders and recompile in watch mode.

```
// Loader adding a header
var path = require("path");
module.exports = function(source) {
  this.cacheable();
  var callback = this.async();
  var headerPath = path.resolve("header.js");
  this.addDependency(headerPath);
  fs.readFile(headerPath, "utf-8", function(err, header) {
    if(err) return callback(err);
    callback(null, header + "\n" + source);
  });
};
```

Resolve dependencies

In many languages there is some schema to specify dependencies. i. e. in css there is `@import` and `url(...)`. These dependencies should be resolved by the module system.

There are two options to do this:

- Transform them to `require`s.
- Use the `this.resolve` function to resolve the path

Example 1 `css-loader`: The `css-loader` transform dependencies to `require`s, by replacing `@import`s with a require to the other stylesheet (processed with the `css-loader` too) and `url(...)` with a `require` to the referenced file.

Example 2 `less-loader`: The `less-loader` cannot transform `@import`s to `require`s, because all less files need to be compiled in one pass to track variables and mixins. Therefore the `less-loader` extends the less compiler with a custom path resolving logic.

This custom logic uses `this.resolve` to resolve the file with the configuration of the module system (aliasing, custom module directories, etc.).

If the language only accept relative urls (like css: `url(file)` always means `./file`), there is the `~`-convention to specify references to modules:

```
url(file) -> require("./file")
url(~module) -> require("module")
```

Extract common code

Don't generate much code that is common in every module processed by that loader. Create a (runtime) file in the loader and generate a `require` to that common code.

Do not embed absolute paths

Don't put absolute paths in to the module code. They break hashing when the root for the project is moved. There is a method `stringifyRequest` in loader-utils which converts an absolute path to an relative one.

Example:

```
var loaderUtils = require("loader-utils");

return "var runtime = require(" +
  loaderUtils.stringifyRequest(this, "!" + require.resolve("module/runtime")) +
  ");";
```

Use a library as `peerDependencies` when they wrap it

Using a peerDependency allows the application developer to specify the exact version in `package.json` if desired. The dependency should be relatively open to allow updating the library without needing to publish a new loader version.

```
"peerDependencies": {
  "library": "^1.3.5"
}
```

Programmable objects as `query`-option

There are situations where your loader requires programmable objects with functions which cannot stringified as `query`-string. The `less-loader`, for example, provides the possibility to specify LESS-plugins. In these cases, a loader is allowed to extend webpack's `options`-object to retrieve that specific option. In order to avoid name collisions, however, it is important that the option is namespaced under the loader's camelCased npm-name.

Example:

```
// webpack.config.js
module.exports = {
  ...
  lessLoader: {
    lessPlugins: [
      new LessPluginCleanCSS({advanced: true})
    ]
  }
};
```

The loader should also allow to specify the config-key (e.g. `lessLoader`) via `query`. See discussion and example implementation.

Plugins expose the full potential of the webpack engine to third-party developers. Using staged build callbacks, developers can introduce their own behaviors into the webpack build process. Building plugins is a bit more advanced than building loaders, because you'll need to understand some of the webpack low-level internals to hook into them. Be prepared to read some source code!

Compiler and Compilation

Among the two most important resources while developing plugins are the `compiler` and `compilation` objects. Understanding their roles is an important first step in extending the webpack engine.

- The `compiler` object represents the fully configured webpack environment. This object is built once upon starting webpack, and is configured with all operational settings including options, loaders, and plugins. When applying a plugin to the webpack environment, the plugin will receive a reference to this compiler. Use the compiler to access the main webpack environment.
- A `compilation` object represents a single build of versioned assets. While running webpack development middleware, a new compilation will be created each time a file change is detected, thus generating a new set of compiled assets. A compilation surfaces information about the present state of module resources, compiled assets, changed files, and watched dependencies. The compilation also provides many callback points at which a plugin may choose to perform custom actions.

These two components are an integral part of any webpack plugin (especially a `compilation`), so developers will benefit by familiarizing themselves with these source files:

- Compiler Source
- Compilation Source

Basic plugin architecture

Plugins are instantiated objects with an `apply` method on their prototype. This `apply` method is called once by the Webpack compiler while installing the plugin. The `apply` method is given a reference to the underlying Webpack compiler, which grants access to compiler callbacks. A simple plugin is structured as follows:

```
function HelloWorldPlugin(options) {
  // Setup the plugin instance with options...
}

HelloWorldPlugin.prototype.apply = function(compiler) {
  compiler.plugin('done', function() {
    console.log('Hello World!');
  });
};

module.exports = HelloWorldPlugin;
```

Then to install the plugin, just include an instance in your webpack config `plugins` array:

```
var HelloWorldPlugin = require('hello-world');

var webpackConfig = {
  // ... config settings here ...
  plugins: [
    new HelloWorldPlugin({options: true})
  ]
};
```

Accessing the compilation

Using the compiler object, you may bind callbacks that provide a reference to each new compilation. These compilations provide callbacks for hooking into numerous steps within the build process.

```
function HelloCompilationPlugin(options) {}

HelloCompilationPlugin.prototype.apply = function(compiler) {

  // Setup callback for accessing a compilation:
  compiler.plugin("compilation", function(compilation) {

    // Now setup callbacks for accessing compilation steps:
    compilation.plugin("optimize", function() {
      console.log("Assets are being optimized.");
    });
  });
};

module.exports = HelloCompilationPlugin;
```

For more information on what callbacks are available on the `compiler`, `compilation`, and other important objects, see the plugins doc.

Async compilation plugins

Some compilation plugin steps are asynchronous, and pass a callback function that *must* be invoked when your plugin is finished running.

```
function HelloAsyncPlugin(options) {}

HelloAsyncPlugin.prototype.apply = function(compiler) {
  compiler.plugin("emit", function(compilation, callback) {

    // Do something async...
    setTimeout(function() {
      console.log("Done with async work...");
      callback();
    }, 1000);

  });
};

module.exports = HelloAsyncPlugin;
```

Example

Once we can latch onto the webpack compiler and each individual compilations, the possibilities become endless for what we can do with the engine itself. We can reformat existing files, create derivative files, or fabricate entirely new assets.

Let's write a simple example plugin that generates a new build file called `filelist.md`; the contents of which will list all of the asset files in our build. This plugin might look something like this:

```
function FileListPlugin(options) {}

FileListPlugin.prototype.apply = function(compiler) {
  compiler.plugin('emit', function(compilation, callback) {
    // Create a header string for the generated file:
    var filelist = 'In this build:\n\n';

    // Loop through all compiled assets,
    // adding a new line item for each filename.
```

```
    for (var filename in compilation.assets) {
      filelist += ('- '+ filename +'\n');
    }

    // Insert this list into the webpack build as a new file asset:
    compilation.assets['filelist.md'] = {
      source: function() {
        return filelist;
      },
      size: function() {
        return filelist.length;
      }
    };

    callback();
  });
};

module.exports = FileListPlugin;
```

Plugins grant unlimited opportunity to perform customizations within the webpack build system. This allows you to create custom asset types, perform unique build modifications, or even enhance the webpack runtime while using middleware. The following are some features of webpack that become useful while writing plugins.

Exploring assets, chunks, modules, and dependencies

After a compilation is sealed, all structures within the compilation may be traversed.

```
function MyPlugin() {}

MyPlugin.prototype.apply = function(compiler) {
  compiler.plugin('emit', function(compilation, callback) {

    // Explore each chunk (build output):
    compilation.chunks.forEach(function(chunk) {
      // Explore each module within the chunk (built inputs):
      chunk.modules.forEach(function(module) {
        // Explore each source file path that was included into the module:
        module.fileDependencies.forEach(function(filepath) {
          // we've learned a lot about the source structure now...
        });
      });

      // Explore each asset filename generated by the chunk:
      chunk.files.forEach(function(filename) {
        // Get the asset source for each file generated by the chunk:
        var source = compilation.assets[filename].source();
      });
    });

    callback();
  });
};

module.exports = MyPlugin;
```

- `compilation.modules` : An array of modules (built inputs) in the compilation. Each module manages the build of a raw file from your source library.
- `module.fileDependencies` : An array of source file paths included into a module. This includes the source JavaScript file itself (ex: `index.js`), and all dependency asset files (stylesheets, images, etc) that it has required. Reviewing dependencies is

useful for seeing what source files belong to a module.

- `compilation.chunks` : An array of chunks (build outputs) in the compilation. Each chunk manages the composition of a final rendered assets.
- `chunk.modules` : An array of modules that are included into a chunk. By extension, you may look through each module's dependencies to see what raw source files fed into a chunk.
- `chunk.files` : An array of output filenames generated by the chunk. You may access these asset sources from the `compilation.assets` table.

Monitoring the watch graph

While running webpack middleware, each compilation includes a `fileDependencies` array (what files are being watched) and a `fileTimestamps` hash that maps watched file paths to a timestamp. These are extremely useful for detecting what files have changed within the compilation:

```
function MyPlugin() {
  this.startTime = Date.now();
  this.prevTimestamps = {};
}

MyPlugin.prototype.apply = function(compiler) {
  compiler.plugin('emit', function(compilation, callback) {

    var changedFiles = Object.keys(compilation.fileTimestamps).filter(function(wat
chfile) {
      return (this.prevTimestamps[watchfile] || this.startTime) < (compilation.fil
eTimestamps[watchfile] || Infinity);
    }.bind(this));

    this.prevTimestamps = compilation.fileTimestamps;
    callback();
  }.bind(this));
};

module.exports = MyPlugin;
```

You may also feed new file paths into the watch graph to receive compilation triggers when those files change. Simply push valid filepaths into the `compilation.fileDependencies` array to add them to the watch. Note: the

`fileDependencies` array is rebuilt in each compilation, so your plugin must push its own watched dependencies into each compilation to keep them under watch.

Changed chunks

Similar to the watch graph, it's fairly simple to monitor changed chunks (or modules, for that matter) within a compilation by tracking their hashes.

```
function MyPlugin() {
  this.chunkVersions = {};
}

MyPlugin.prototype.apply = function(compiler) {
  compiler.plugin('emit', function(compilation, callback) {

    var changedChunks = compilation.chunks.filter(function(chunk) {
      var oldVersion = this.chunkVersions[chunk.name];
      this.chunkVersions[chunk.name] = chunk.hash;
      return chunk.hash !== oldVersion;
    }.bind(this));

    callback();
  }.bind(this));
};

module.exports = MyPlugin;
```

Pull requests into `master`

When you land commits on your `master` branch, select the *Create Merge-Commit* option.

Cut a release

```
npm version patch && git push --follow-tags && npm publish
npm version minor && git push --follow-tags && npm publish
npm version major && git push --follow-tags && npm publish
```

*This will increment the package version, commits the changes, cuts a **local tag**, push to github & publish the npm package.*

After that go to the github releases page and write a Changelog for the new tag.

webpack enables use of loaders to preprocess files. This allows you to bundle any static resource way beyond JavaScript. You can easily write your own loaders using Node.js.

Loaders are activated by using `loadername!` prefixes in `require()` statements, or are automatically applied via regex from your webpack configuration – see configuration.

Files

- `raw-loader` Loads raw content of a file (utf-8)
- `val-loader` Executes code as module and consider exports as JS code
- `url-loader` Works like the file loader, but can return a data URL if the file is smaller than a limit
- `file-loader` Emits the file into the output folder and returns the (relative) URL

JSON

- `json-loader` Loads a JSON file (included by default)
- `json5-loader` Loads and transpiles a JSON 5 file
- `cson-loader` Loads and transpiles a CSON file

Transpiling

- `script-loader` Executes a JavaScript file once in global context (like in script tag), requires are not parsed
- `babel-loader` Loads ES2015+ code and transpiles to ES5 using Babel
- `traceur-loader` Loads ES2015+ code and transpiles to ES5 using Traceur
- `typescript-loader` Loads TypeScript like JavaScript
- `coffee-loader` Loads CoffeeScript like JavaScript

Templating

- `html-loader` Exports HTML as string, require references to static resources
- `pug-loader` Loads Pug templates and returns a function
- `jade-loader` Loads Jade templates and returns a function
- `markdown-loader` Compiles Markdown to HTML
- `posthtml-loader` Loads and transforms a HTML file using PostHTML

- `handlebars-loader` Compiles Handlebars to HTML

Styling

- `style-loader` Add exports of a module as style to DOM
- `css-loader` Loads CSS file with resolved imports and returns CSS code
- `less-loader` Loads and compiles a LESS file
- `sass-loader` Loads and compiles a SASS/SCSS file
- `stylus-loader` Loads and compiles a Stylus file
- `postcss-loader` Loads and transforms a CSS/SSS file using PostCSS

Linting && Testing

- `mocha-loader` Tests with mocha (Browser/NodeJS)
- `eslint-loader` PreLoader for linting code using ESLint
- `jshint-loader` PreLoader for linting code using JSHint
- `jscs-loader` PreLoader for code style checking using JSCS
- `coverjs-loader` PreLoader to determine the testing coverage using CoverJS

Frameworks

- `vue-loader` Loads and compiles Vue Components
- `polymer-loader` Process HTML & CSS with preprocessor of choice and `require()` Web Components like first-class modules
- `angular2-template-loader` Loads and compiles Angular Components

For more third-party loaders, see the list from awesome-webpack.

webpack provides flexible and powerful customization api in the form of plugins. Using plugins, we can plug functionality into webpack. Additionally, webpack provides lifecycle hooks into which plugins can be registered. At each of these lifecycle points, webpack will run all of the registered plugins and provide them with the current state of the webpack compilation.

Tapable & Tapable instances

The plugin architecture is mainly possible for webpack due to an internal library named `Tapable`. **Tapable Instances** are classes in the webpack source code which have been extended or mixed in from class `Tapable`.

For plugin authors, it is important to know which are the `Tapable` instances in the webpack source code. These instances provide a variety of event hooks into which custom plugins can be attached. Hence, throughout this section are a list of all of the webpack `Tapable` instances (and their event hooks), which plugin authors can utilize.

For more information on `Tapable` visit the tapable repository or visit the complete overview

Creating a Plugin

A plugin for `webpack` consists of

- A named JavaScript function.
- Defines `apply` method in it's prototype.
- Specifies webpack's event hook to attach itself.
- Manipulates webpack internal instance specific data.
- Invokes webpack provided callback after functionality is complete.

```
// A named JavaScript function.
function MyExampleWebpackPlugin() {

};

// Defines `apply` method in it's prototype.
MyExampleWebpackPlugin.prototype.apply = function(compiler) {
  // Specifies webpack's event hook to attach itself.
  compiler.plugin('webpacksEventHook', function(compilation /* Manipulates webpack
 internal instance specific data. */, callback) {
    console.log("This is an example plugin!!!");
```

```
    // Invokes webpack provided callback after functionality is complete.
    callback();
  });
};
```

Different Plugin Shapes

A plugin can be classified into types based on the event it is registered to. Every event hook decides how it is going to apply the plugins in its registry.

- **synchronous** The Tapable instance applies plugins using

`applyPlugins(name: string, args: any...)`

`applyPluginsBailResult(name: string, args: any...)`

This means that each of the plugin callbacks will be invoked one after the other with the specific `args`. This is the simplest format for a plugin. Many useful events like `"compile"`, `"this-compilation"` expect plugins to have synchronous execution.

- **waterfall** Plugins applied using

`applyPluginsWaterfall(name: string, init: any, args: any...)`

Here each of the plugins are called one after the other with the args from the return value of the previous plugin. The plugin must take into consider the order of its execution. It must accept arguments from the previous plugin that was executed. The value for the first plugin is `init`. This pattern is used in the Tapable instances which are related to the `webpack` templates like `ModuleTemplate`, `ChunkTemplate` etc.

- **asynchronous** When all the plugins are applied asynchronously using

`applyPluginsAsync(name: string, args: any..., callback: (err?: Error) -> void)`

The plugin handler functions are called with all args and a callback function with the signature `(err?: Error) -> void`. The handler functions are called in order of registration. `callback` is called after all the handlers are called. This is also a commonly used pattern for events like `"emit"`, `"run"`.

- **async waterfall** The plugins will be applied asynchronously in the waterfall manner.

`applyPluginsAsyncWaterfall(name: string, init: any, callback: (err: Error, result: any) -> void)`

The plugin handler functions are called with the current value and a callback function with the signature `(err: Error, nextValue: any) -> void.` When called `nextValue` is the current value for the next handler. The current value for the first handler is `init`. After all handlers are applied, callback is called with the last value. If any handler passes a value for `err`, the callback is called with this error and no more handlers are called. This plugin pattern is expected for events like `"before-resolve"` and `"after-resolve"`.

- **async series** It is the same as asynchronous but if any of the plugins registered fails, then no more plugins are called.

`applyPluginsAsyncSeries(name: string, args: any..., callback: (err: Error, result: any) -> void)`

-**parallel** -

`applyPluginsParallel(name: string, args: any..., callback: (err?: Error) -> void)`

`applyPluginsParallelBailResult(name: string, args: any..., callback: (err: Error, result: any) -> void)`

The `Compiler` module of webpack is the main engine that creates a compilation instance with all the options passed through webpack CLI or `webpack` api or webpack configuration file.

It is exported by `webpack` api under `webpack.Compiler`.

The compiler is used by webpack by instantiating it and then calling the `run` method. Below is a trivial example of how one might use the `Compiler`. In fact, this is really close to how webpack itself uses it.

compiler-example

```
// Can be imported from webpack package
import {Compiler} from 'webpack';

// Create a new compiler instance
const compiler = new Compiler();

// Populate all required options
compiler.options = {...};

// Creating a plugin.
class LogPlugin {
    apply (compiler) {
        compiler.plugin('should-emit', compilation => {
            console.log('should i emit?');
            return true;
        })
    }
}

// Apply the compiler to the plugin
new LogPlugin().apply(compiler);

/* Add other supporting plugins */

// Callback to be executed after run is complete
const callback = (err, stats) => {
    console.log('Compiler has finished execution.');
    /* Do something to show the stats */
};

// call run on the compiler along with the callback
compiler.run(callback);
```

The `Compiler` is what we call a `Tapable` instance. By this, we mean that it mixes in `Tapable` class to imbibe functionality to register and call plugins on itself. Most user facing plugins are first registered on the `Compiler`. The working of a Compiler can be condensed into the following highlights

- Usually there is one master instance of Compiler. Child compilers can be created for delegating specific tasks.
- A lot of the complexity in creating a compiler goes into populating all the relevant options for it.
- `webpack` has `WebpackOptionsDefaulter` and `WebpackOptionsApply` specifically designed to provide the `Compiler` with all the intial data it requires.
- The `Compiler` is just ultimately just a function which performs bare minimum functionality to keep a lifecycle running. It delegates all the loading/bundling/writing work to various plugins.
- `new LogPlugin(args).apply(compiler)` registers the plugin to any particular hook event in the `Compiler` 's lifecycle.
- The `Compiler` exposes a `run` method which kickstarts all compilation work for `webpack`. When that is done, it should call the passed in `callback` function. All the tail end work of logging stats and errors are done in this callback function.

Watching

However, the `Compiler` supports two flavors of execution. One on watch mode and one on a normal single run. While it essentially performs the same functionality while watching, there are some additions to the lifecycle events. This allows `webpack` to have Watch specific plugins.

MultiCompiler

This module, MultiCompiler, allows webpack to run multiple configurations in separate compiler. If the `options` parameter in the webpack's NodeJS api is an array of options, webpack applies separate compilers and calls the `callback` method at the end of each compiler execution.

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

var config1 = {
  entry: './index1.js',
```

```
  output: {filename: 'bundle1.js'}
}
var config2 = {
  entry: './index2.js',
  output: {filename:'bundle2.js'}
}

webpack([config1, config2], (err, stats) => {
  process.stdout.write(stats.toString() + "\n");
})
```

Event Hooks

This a reference guide to all the event hooks exposed by the `Compiler`.

Event name	Reason	Params	Type
`entry-option`	-	-	bailResult
`after-plugins`	After setting up initial set of plugins	`compiler`	sync
`after-resolvers`	After setting up the resolvers	`compiler`	sync
`environment`	-	-	sync
`after-environment`	Environment setup complete	-	sync
`before-run`	`compiler.run()` starts	`compiler`	async
`run`	Before reading records	`compiler`	async
`watch-run`	Before starting compilation after watch	`compiler`	async
`normal-module-factory`	After creating a `NormalModuleFactory`	`normalModuleFactory`	sync
`context-module-factory`	After creating a `ContextModuleFactory`	`contextModuleFactory`	sync
`before-compile`	Compilation parameters created	`compilationParams`	sync
	Before creating new		

	compilation		
`this-compilation`	Before emitting `compilation` event	`compilation`	sync
`compilation`	Compilation creation completed	`compilation`	sync
`make`		`compilation`	parallel
`after-compile`		`compilation`	async
`should-emit`	Can return true/false at this point	`compilation`	bailResult
`need-additional-pass`		-	bailResult
`emit`	Before writing emitted assets to output dir	`compilation`	async
`after-emit`	After writing emitted assets to output dir	`compilation`	async
`done`	Completion of compile	`stats`	sync
`fail`	Failure of compile	`error`	sync
`invalid`	After invalidating a watch compile	`fileName`, `changeTime`	sync

Examples

?> Adds examples of usage for some of the above events

> TODO

> TODO

> TODO

> TODO

> TODO

Tapable is small library that allows you to add and apply plugins to a javascript module. It can be inherited or mixed in to other modules. It is similar to NodeJS's `EventEmitter` class, focusing on custom event emission and manipulation. However, in addition to this, `Tapable` allows you to have access to the "emittee" or "producer" of the event through callbacks arguments.

`Tapable` has four groups of member functions:

- `plugin(name:string, handler:function)` - This allows a custom plugin to register into a **Tapable instance**'s event. This acts as the same as `on()` of `EventEmitter`, for registering a handler/listener to do something when the signal/event happens.
- `apply(…pluginInstances: (AnyPlugin|function)[])` - `AnyPlugin` should be subclass of AbstractPlugin, or a class (or object, rare case) has an `apply` method, or just a function with some registration code inside. This method is just to **apply** plugins' definition, so that the real event listeners can be registered into the **Tapable instance**'s registry.
- `applyPlugins*(name:string, …)` - The **Tapable instance** can apply all the plugins under a particular hash using these functions. These group of method act like `emit()` of `EventEmitter`, to control the event emission meticulously with various strategy for various use cases.
- `mixin(pt: Object)` - a simple method to extend `Tapable`'s prototype as a mixin rather than inheritance.

The different `applyPlugins*` methods cover the following use cases:

- Plugins can run serially
- Plugins can run in parallel
- Plugins can run one after the other but taking input from the previous plugin (waterfall)
- Plugins can run asynchronously
- Quit runing plugins on bail: that is once one plugin returns non-`undefined`, jump out of the run flow and return *the return of that plugin*. This sounds like `once()` of `EventEmitter` but is totally different.

Example

One of webpack's **Tapable instances**, Compiler, is responsible for compiling the webpack configuration object and returning a Compilation instance. When the Compilation instance runs, it creates the required bundles.

See below is a simplified version of how this looks using `Tapable`.

node_modules/webpack/lib/Compiler.js

```
var Tapable = require("tapable");
function Compiler() {
    Tapable.call(this);
}
Compiler.prototype = Object.create(Tapable.prototype);
```

Now to write a plugin on the compiler,

my-custom-plugin.js

```
function CustomPlugin() {}
CustomPlugin.prototype.apply = function(compiler) {
    compiler.plugin('emit', pluginFunction);
}
```

The compiler executes the plugin at the appropriate point in its lifecycle by

node_modules/webpack/lib/Compiler.js

```
this.apply*("emit",options) // will fetch all plugins under 'emit' name and run th
em.
```

MainTemplate

HotUpdateChunkTemplate

ChunkTemplate

ModuleTemplate

FunctionModuleTemplate

Event Hooks

Examples