Webpack 4: практические рекомендации по настройке

* [Переводы](https://tproger.ru/category/translations/), 8 мая 2018 в 16:44



Перевод статьи [«A tale of Webpack 4 and how to finally configure it in the right way»](https://hackernoon.com/a-tale-of-webpack-4-and-how-to-finally-configure-it-in-the-right-way-4e94c8e7e5c1)

*За перевод благодарим коллектив ~~[реакт.рф](http://xn--80ajqvn.xn--p1ai/" \t "_blank)~~ и автора оригинальной англоязычной версии статьи, Маргариту Образцову.*

В Сети есть огромное количество туториалов, поэтому вы, вероятно, видели тысячи различных способов конфигурирования файла Webpack. Каждый из них работает и может служить примером, однако при этом все они немного разные. Почему так? Сам Webpack развивается очень быстро, и многие загрузчики (лоадеры) и плагины должны идти с ним в ногу. Это основная причина, по которой конфиги так отличаются друг от друга: с различными комбинациями одних и тех же инструментов все может как работать, так и не работать.

Позвольте мне выразить свое искреннее мнение: многие люди жалуются на Webpack и трудности в конфигурации. Во многих случаях это так, хотя с моим опытом работы с gulp и grunt следует сказать, что вы сталкиваетесь с теми же проблемами несовместимости, когда используете npm-модули.

Webpack сейчас является самым популярным бандлером, у которого недавно произошли серьезные обновления. Они предполагают много нового: нулевую конфигурацию, разумные значения по умолчанию, улучшенную производительность, готовые инструменты оптимизации.

**[Конференция HolyJS 2020 Piter](https://tproger.ru/events/holyjs-2020-piter/?utm_source=in_text)**

[10–11 апреля, Москва, 10 750–138 000 ₽](https://tproger.ru/events/holyjs-2020-piter/?utm_source=in_text)

[tproger.ru](https://tproger.ru/events/holyjs-2020-piter/?utm_source=in_text)

[События и курсы на tproger.ru](https://tproger.ru/events/)

Если вы совершенно незнакомы с Webpack, для начала неплохо будет ознакомиться с [документацией](https://webpack.js.org/concepts/). У Webpack она довольно хорошая, с изложением в нескольких частях, поэтому в этом посте я быстро пробегу по основным понятиям.

**Нулевая конфигурация:**Webpack 4 не требует наличия конфигурационного файла, это одно из новшеств 4-ой версии. Настройка Webpack работает шаг за шагом, поэтому нет необходимости выполнять масштабную конфигурацию с самого начала.

**Улучшенная производительность:** Webpack 4 — самая быстрая версия Webpack.

**Разумные значения по умолчанию:** основные понятия Webpack 4 — точка входа, точка вывода, загрузчики (лоадеры) и плагины. Я не буду подробно их описывать, только скажу, что между загрузчиками и плагинами очень расплывчатая разница. Все зависит от того, как автор реализовал библиотеку.

Основные понятия

Точка входа (entry)

Это ваш .js файл. Вы, возможно, видели несколько конфигураций, в которые включены файлы .scss или .css. Это довольно серьезный хак, который может привести ко множеству неожиданных ошибок. Также иногда встречается вход с несколькими .js файлами. Хотя некоторые плагины позволяют это сделать, скажу, что так делать стоит только тогда, когда вы четко понимаете, зачем вам это нужно.

Точка вывода (output)

Это ваша build/ или dist/ или wateveryounameit/ папка, где будет размещен конечный .js файл. Это ваш окончательный результат, который впоследствии будет загружен в index.html.

Загрузчики (loaders)

В основном компилируют или транспилируют ваш код, например, postcss-loader будет проводить ваши стили через разные плагины.

Плагины (plugins)

Играют жизненно важную роль в выводе кода в файлы.

Быстрый старт

Создайте новую директорию и зайдите в нее:

mkdir webpack-4-tutorial

cd webpack-4-tutorial

Инициализируем package.json:

npm init

Необходимо загрузить Webpack v4 в качестве модуля и webpack-cli, чтобы запустить его из терминала.

npm install webpack webpack-cli --save-dev

Убедитесь, что установлена именно 4 версия, если же нет, вы можете напрямую выбрать ее в вашем файле package.json. Далее откройте package.json и добавьте build-скрипт:

"scripts": {

"dev": "webpack"

}

Скорее всего, вылезет предупреждение:

 WARNING in configuration

The ‘mode’ option has not been set, webpack will fallback to ‘production’ for this value. Set ‘mode’ option to ‘development’ or ‘production’ to enable defaults for each environment.

You can also set it to ‘none’ to disable any default behavior. Learn more: https://webpack.js.org/concepts/mode/

Поэтому следующим шагом мы установим флаг режима.

Режимы Webpack 4 (Webpack modes)

Вам необходимо отредактировать скрипт, чтобы в нем появился флаг режима (mode flag):

"scripts": {

"dev": "webpack --mode development"

}

ERROR in Entry module not found: Error: Can’t resolve './src' in '~/webpack-4-quickstart'

Это значит, что Webpack ищет папку ./src/ с файлом index.js.  Давайте создадим директорию с .js файлом, например ./src/index.js, и добавьте в него такой код:

console.log("hello, world");

Теперь запустите dev-скрипт:

npm run dev

Теперь у вас есть директория ./dist/main.js. Отлично, так как мы знаем, что код скомпилирован. Но что произошло?

Новый Webpack не требует наличия конфигурации, поэтому перед началом использования вам не нужно возиться с webpack.config.js. Из-за этого Webpack вынужден действовать по умолчанию, и он будет искать папку ./src, а в ней index.js и выводить в ./dist/main.js; main.js — это ваш скомпилированный файл с зависимостями.

Наличие двух конфигурационных файлов — распространенная практика в Webpack, особенно в крупных проектах. Обычно один файл используется для разработки, другой — для продакшена. В Webpack есть два режима: продакшен и разработка. Это избавляет от необходимости создавать два конфигурационных файла (для средних по размеру проектов).

"scripts": {

"dev": "webpack --mode development",

"build": "webpack --mode production"

}

Если вы сейчас проверите файл main.js, то увидите, что он не изменился.

Примечание В данном примере я буду использовать build-скрипт, поскольку он уже “из коробки” обеспечивает множеством инструментов оптимизации и с этого момента можно свободно их использовать. Основное различие между скриптами build и dev в том, как они выводят файлы. Build создан для продакшен-кода, dev — для разработки: он поддерживает «горячую» замену модулей, webpack-dev-server и многие другие вещи, которые помогают разработчикам.

Вы легко можете переопределять значения по умолчанию в скриптах npm, просто используя флаги:

"scripts": {

"dev": "webpack --mode development ./src/index.js --output ./dist/main.js",

"build": "webpack --mode production ./src/index.js --output ./dist/main.js"

}

Это перепишет опции по умолчанию без каких-либо других настроек.

В качестве упражнения попробуйте также флаги:

* флаг наблюдения watch для включения режима просмотра. Он будет следить за изменениями вашего файла и компилировать всякий раз при его изменении:

"scripts": {

"dev": "webpack --mode development ./src/index.js --output ./dist/main.js --watch",

"build": "webpack --mode production ./src/index.js --output ./dist/main.js --watch"

}

* флаг входа. Работает так же, как и выход, но перезаписывает точку входа.

Транспиляция вашего кода .js

Современный JS-код в основном написан на ES6, а ES6 поддерживается не всеми браузерами. Поэтому вам необходимо транспилировать его. Транспиляция — волшебное слово для перевода ES6 в ES5. Для этого можно использовать babel — наиболее популярный инструмент для транспиляции. Разумеется, он используется не только для ES6, но и для многих инструментов JS, таких как TypeScript, React и т.д.

npm install babel-core babel-loader babel-preset-env --save-dev

Далее нужно создать файл конфигурации для babel:

nano .babelrc

Вставляем туда:

{

"presets": [

"env"

]

}

Теперь у нас есть два варианта конфигурации babel-загрузчика:

1. Использовать конфигурационный файл webpack.config.js.
2. Использовать --module-bind в npm-скриптах.

Технически вы многое можете сделать с новыми флагами, представленными в Webpack, однако для простоты я бы предпочла webpack.config.js.

Конфигурационный файл

На этом этапе мы создадим webpack.config.js со следующим содержимым:

// Webpack v4

const path = require('path');

module.exports = {

entry: { main: './src/index.js' },

output: {

path: path.resolve(\_\_dirname, 'dist'),

filename: 'main.js'

},

module: {

rules: [

{

test: /\.js$/,

exclude: /node\_modules/,

use: {

loader: "babel-loader"

}

}

]

}

}

Также теперь мы удалим флаги из наших npm-скриптов:

"scripts": {

"build": "webpack --mode production",

"dev": "webpack --mode development"

},

Теперь, когда мы запускаем npm run build, он должен вывести нам небольшой js-файл в ./dist/main.js. Если это не происходит, попробуйте переустановить babel-загрузчик.

Наиболее распространенным паттерном Webpack является его использование для компиляции приложения React. Несмотря на это, мы не будем зацикливаться на React’е, поскольку я хочу, чтобы структура была как можно более независимой от фреймворков. Вместо этого я покажу вам, как обрабатывать и создавать конфиги для .html и .css.

Импорт HTML и CSS

Для начала в нашей папке ./dist создадим небольшой файл index.html.

<html>

<head>

<link rel="stylesheet" href="style.css">

</head>

<body>

<div>Hello, world!</div>

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

</body>

</html>

Как видите, здесь мы импортируем style.css. Давайте настроим его! Как я уже упомянула, для Webpack у нас есть только одна точка входа. Так куда же мы поставим наш css?

Создаем style.css в нашей папке ./src:

div {

color: red;

}

Не забываем включить это в js-файл:

import "./style.css";

console.log("hello, world");

В Webpack создайте новое правило для файлов css:

// Webpack v4

const path = require('path');

const ExtractTextPlugin = require('extract-text-webpack-plugin');

module.exports = {

entry: { main: './src/index.js' },

output: {

path: path.resolve(\_\_dirname, 'dist'),

filename: 'main.js'

},

module: {

rules: [

{

test: /\.js$/,

exclude: /node\_modules/,

use: {

loader: "babel-loader"

}

},

{

test: /\.css$/,

use: ExtractTextPlugin.extract(

{

fallback: 'style-loader',

use: ['css-loader']

})

}

]

}

};

В терминале запустите:

npm install extract-text-webpack-plugin --save-dev

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

Необходимо использовать ExtractTextPlugin для компиляции нашего CSS. Как видите, для .css мы также добавили новое правило.

Краткое описание обычной работы правил:

{

test: /\.YOUR\_FILE\_EXTENSION$/, // расширение файла

exclude: /SOMETHING THAT IS THAT EXTENSION BUT SHOULD NOT BE PROCESSED/, // что-то, что не должно обрабатываться

use: {

loader: "loader for your file extension or a group of loaders" // загрузчик или группа загрузчиков

}

}

Примечание Нам необходимо использовать ExtractTextPlugin, потому что Webpack по умолчанию понимает только формат .js. ExtractTextPlugin получает ваш .css и извлекает его в отдельный css-файл в вашем каталоге ./dist.

Спойлер: в некоторых статьях пишут, что ExtractTextPlugin не работает с Webpack 4, но у меня он работал 🙂 Это доказывает мою точку зрения о неоднородном поведении модулей при настройке. Если же у вас этот модуль не работает, вы можете переключиться на MiniCssExtractPlugin. Далее в этой статье я покажу вам, как настроить и его тоже.

// Webpack v4

const path = require('path');

const ExtractTextPlugin = require('extract-text-webpack-plugin');

module.exports = {

entry: { main: './src/index.js' },

output: {

path: path.resolve(\_\_dirname, 'dist'),

filename: 'main.js'

},

module: {

rules: [

{

test: /\.js$/,

exclude: /node\_modules/,

use: {

loader: "babel-loader"

}

},

{

test: /\.css$/,

use: ExtractTextPlugin.extract(

{

fallback: 'style-loader',

use: ['css-loader']

})

}

]

},

plugins: [

new ExtractTextPlugin({filename: 'style.css'})

]

};

У Webpack, начиная с 4 версии, есть проблемы с этим плагином, поэтому вы можете столкнуться с [определенной ошибкой](https://github.com/webpack-contrib/extract-text-webpack-plugin/issues/70).

Чтобы ее исправить, запустите:

npm install -D extract-text-webpack-plugin@next

Совет: гуглите вылезающие ошибки и ищите подобные вопросы в GitHub или просто задайте свой вопрос на StackOverflow.

Затем ваш css код должен скомпилироваться в ./dist/style.css.

На данном этапе dev-зависимости в моем package.json выглядят так:

"devDependencies": {

"babel-core": "^6.26.0",

"babel-loader": "^7.1.4",

"babel-preset-env": "^1.6.1",

"css-loader": "^0.28.11",

"extract-text-webpack-plugin": "^4.0.0-beta.0",

"style-loader": "^0.20.3",

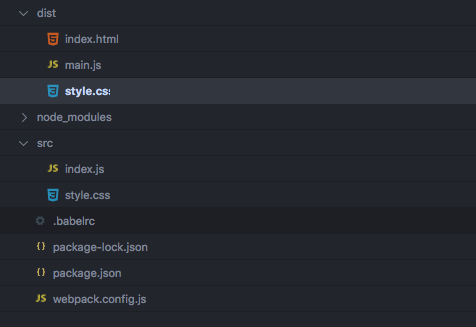
"webpack": "^4.4.1",

"webpack-cli": "^2.0.12"

}

Обратите внимание, что иная комбинация может не работать, поскольку даже обновление webpack-cli v2.0.12 до 2.0.13 может «сломать» ее. #justwebpackthings

Теперь ваш style.css должен выводиться в папку ./dist:

[](https://s3.tproger.ru/uploads/2018/04/picpicpic.png)

Поддержка SCSS

Очень часто разрабатываются веб-сайты с [SASS](https://ru.wikipedia.org/wiki/Sass) и [PostCSS](https://ru.wikipedia.org/wiki/PostCSS" \t "_blank). Они весьма полезны. Поэтому мы сперва включим поддержку SASS. Переименуем наш ./src/style.css и создадим другую папку для хранения файлов .scss. Теперь необходимо добавить поддержку форматирования .scss.

npm install node-sass sass-loader --save-dev

Замените style.scss на ./scss/main.scss в файле .js.

Шаблон HTML

Теперь давайте создадим html-шаблон файла. Добавьте файл index.html в ./src с точно такой же структурой.

<html>

<head>

<link rel="stylesheet" href="style.css">

</head>

<body>

<div>Hello, world!</div>

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

</body>

</html>

Чтобы использовать этот файл в качестве шаблона, нам понадобится html-плагин.

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

Добавьте его в ваш вэбпак-файл:

// Webpack v4

const path = require('path');

const ExtractTextPlugin = require('extract-text-webpack-plugin');

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

module.exports = {

entry: { main: './src/index.js' },

output: {

path: path.resolve(\_\_dirname, 'dist'),

filename: 'main.js'

},

module: {

rules: [

{

test: /\.js$/,

exclude: /node\_modules/,

use: {

loader: "babel-loader"

}

},

{

test: /\.scss$/,

use: ExtractTextPlugin.extract(

{

fallback: 'style-loader',

use: ['css-loader', 'sass-loader']

})

}

]

},

plugins: [

new ExtractTextPlugin(

{filename: 'style.css'}

),

new HtmlWebpackPlugin({

inject: false,

hash: true,

template: './src/index.html',

filename: 'index.html'

})

]

};

Теперь ваш файл из ./src/index.html стал шаблоном для конечного файла index.html.

Чтобы убедиться, что все работает, удалите все файлы из папки ./dist и саму папку. Затем запустите dev-скрипт:

rm -rf ./dist

npm run dev

Вы увидите, что появилась папка ./dist, и в ней 3 файла: index.html, style.css, script.js.

Кэширование и хеширование

Одна из наиболее распространенных проблем в разработке — реализация кэширования. Очень важно понять, как это работает, ведь вы же хотите, чтобы у ваших пользователей всегда была последняя версия кода.

Поскольку этот пост затрагивает главным образом конфигурацию Webpack, — одним из самых популярных способов решения проблем кеширования является добавление хеш-номера в активные (находящиеся в разработке) файлы (asset), такие как style.css и script.js. Об этом можно почитать [здесь](https://developers.google.com/web/fundamentals/performance/webpack/use-long-term-caching#split-the-code-into-routes-and-pages). Хеширование необходимо, чтобы браузер «научился» запрашивать только измененные файлы.

Webpack 4 имеет встроенные функции, реализованные через [chunkhash](https://webpack.js.org/guides/caching/" \t "_blank). Это можно сделать следующим образом:

// Webpack v4

const path = require('path');

const ExtractTextPlugin = require('extract-text-webpack-plugin');

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

module.exports = {

entry: { main: './src/index.js' },

output: {

path: path.resolve(\_\_dirname, 'dist'),

filename: '[name].[chunkhash].js'

},

module: {

rules: [

{

test: /\.js$/,

exclude: /node\_modules/,

use: {

loader: "babel-loader"

}

},

{

test: /\.scss$/,

use: ExtractTextPlugin.extract(

{

fallback: 'style-loader',

use: ['css-loader', 'sass-loader']

})

}

]

},

plugins: [

new ExtractTextPlugin(

{filename: 'style.[chunkhash].css', disable: false, allChunks: true}

),

new HtmlWebpackPlugin({

inject: false,

hash: true,

template: './src/index.html',

filename: 'index.html'

}),

]

};

В файл ./src/index.html добавьте:

<html>

<head>

<link rel="stylesheet" href="<%=htmlWebpackPlugin.files.chunks.main.css %>">

</head>

<body>

<div>Hello, world!</div>

<script src="<%= htmlWebpackPlugin.files.chunks.main.entry %>"></script>

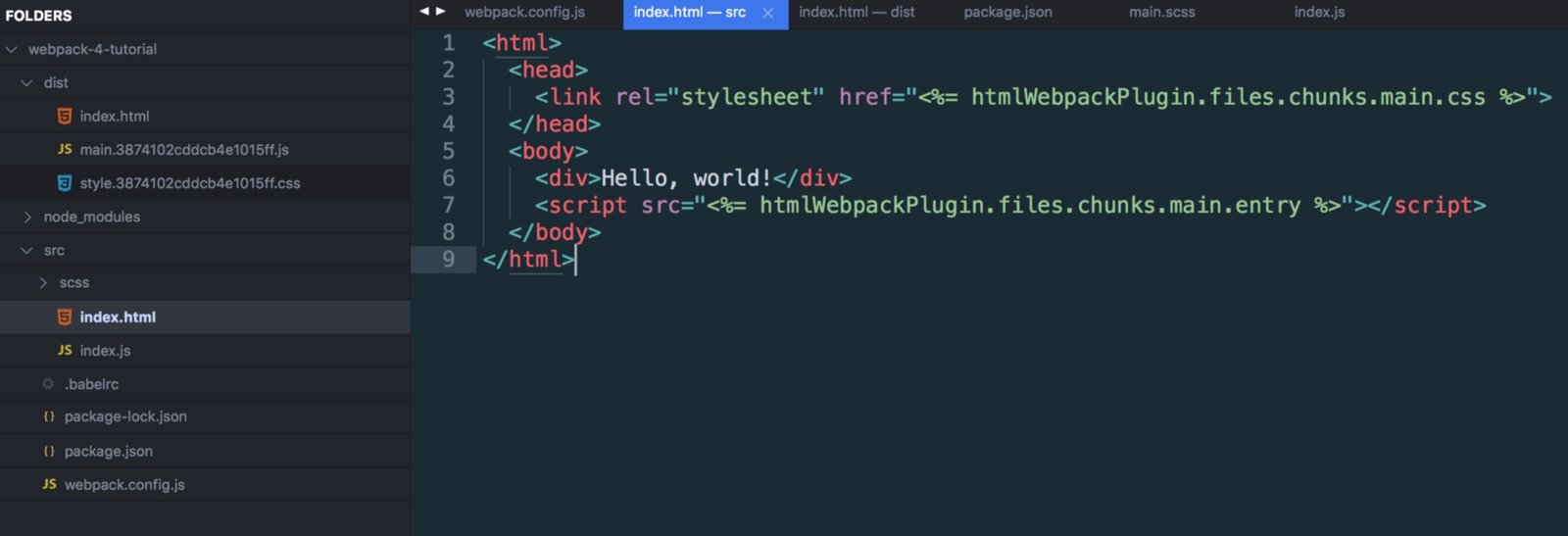
</body>

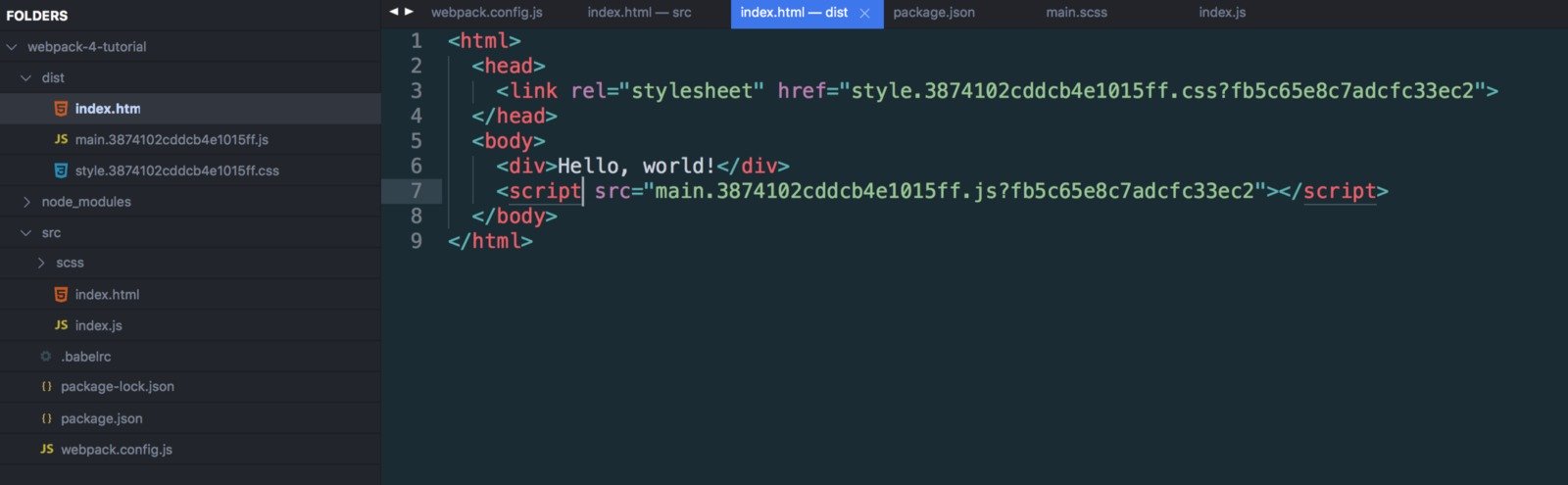
</html>

С помощью такого синтаксиса ваш шаблон «научится» использовать хешированные файлы. Это новая фича, реализованная после возникновения [этой проблемы](https://github.com/jantimon/html-webpack-plugin/pull/14).

Мы будем использовать описанный там htmlWebpackPlugin.files.chunks.main.

Теперь в нашей папке ./dist есть файл index.html:

[](https://s3.tproger.ru/uploads/2018/04/razkartinka.jpg)

[](https://s3.tproger.ru/uploads/2018/04/dvakartinka.jpg)

Далее, если мы ничего не изменим в файлах .js и .css и запустим:

npm run dev

То увидим, что, независимо от количества запусков, числа в хешах должны быть идентичны в обоих файлах.

Проблемы с хешированием и их решения

Хотя у нас есть рабочий алгоритм, он пока не идеален.  
Что, если изменить код в файле .scss? Окей, меняем .scss и снова запускаем dev-скрипт. Новый хеш не генерируется.

Что, если мы добавим строки console.log в наш js-файл вот так:

import "./style.css";

console.log("hello, world");

console.log("Hello, world 2");

Если вы снова запустите dev-скрипт, увидите, что в обоих файлах обновлен хеш.

Эта известная проблема, и этот вопрос даже разобран на [StackOveflow](https://stackoverflow.com/questions/44491064/updating-chunkhash-in-both-css-and-js-file-in-webpack" \t "_blank).

Как теперь это исправить?

Перепробовав немало плагинов, якобы решающих эту проблему, я, наконец, пришла к двум типам решений:

**Решение 1**

Замените [chukhash] на простой [hash] в .css. Это было одно из решений [проблемы](https://github.com/webpack-contrib/extract-text-webpack-plugin/issues/763). Это похоже на конфликт с [webpack 4.3](https://github.com/webpack/webpack/releases/tag/v4.3.0" \t "_blank), в котором введена собственная переменная contenthash. Используйте его вместе с плагином [webpack-md5-hash](https://www.npmjs.com/package/webpack-md5-hash).

Теперь, когда вы вносите изменения в свой файл main.scss и запускаете скрипт dev, с новым хешем должен быть сгенерирован только новый style.css.

// webpack v4

const path = require('path');

const ExtractTextPlugin = require('extract-text-webpack-plugin');

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

const WebpackMd5Hash = require('webpack-md5-hash');

module.exports = {

entry: { main: './src/index.js' },

output: {

path: path.resolve(\_\_dirname, 'dist'),

filename: '[name].[chunkhash].js'

},

module: {

rules: [

{

test: /\.js$/,

exclude: /node\_modules/,

use: {

loader: "babel-loader"

}

},

{

test: /\.scss$/,

use: ExtractTextPlugin.extract(

{

fallback: 'style-loader',

use: ['css-loader', 'sass-loader']

})

}

]

},

plugins: [

new ExtractTextPlugin(

{filename: 'style.[hash].css', disable: false, allChunks: true}

),

new HtmlWebpackPlugin({

inject: false,

hash: true,

template: './src/index.html',

filename: 'index.html'

}),

new WebpackMd5Hash()

]

};

Далее протестируем js-файлы. Теперь оба файла меняют хеш.

**Решение 2**

Все еще могут остаться некоторые конфликты, поэтому теперь попробуем подключить плагин [mini-css-extract](https://github.com/webpack-contrib/mini-css-extract-plugin" \t "_blank).

Плагин Mini-CSS

Плагин Mini CSS предназначен для замены плагина extract-text и в дальнейшем для обеспечения лучшей совместимости.

Я реструктурировала свой Webpack-файл, чтобы скомпилировать style.css с [mini-css-extract-plugin](https://github.com/webpack-contrib/mini-css-extract-plugin" \t "_blank), и у меня все работает.

// webpack v4

const path = require('path');

// const ExtractTextPlugin = require('extract-text-webpack-plugin');

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

const WebpackMd5Hash = require('webpack-md5-hash');

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

module.exports = {

entry: { main: './src/index.js' },

output: {

path: path.resolve(\_\_dirname, 'dist'),

filename: '[name].[chunkhash].js'

},

module: {

rules: [

{

test: /\.js$/,

exclude: /node\_modules/,

use: {

loader: "babel-loader"

}

},

{

test: /\.scss$/,

use: [ 'style-loader', MiniCssExtractPlugin.loader, 'css-loader', 'sass-loader']

}

]

},

plugins: [

// new ExtractTextPlugin(

// {filename: 'style.[hash].css', disable: false, allChunks: true }

// ),

new MiniCssExtractPlugin({

filename: 'style.[contenthash].css',

}),

new HtmlWebpackPlugin({

inject: false,

hash: true,

template: './src/index.html',

filename: 'index.html'

}),

new WebpackMd5Hash()

]

};

Теперь, когда я редактирую main.scss, генерируется новый хеш для style.css. Также, когда я редактирую css, меняются только хеши css, и, когда я редактирую ./src/script.js, меняются только хеши script.js!

Интегрирование PostCSS

Чтобы выходной css-файл был безупречным, мы можем добавить наверх PostCSS.

[PostCSS](https://github.com/postcss/postcss) предоставляет вам autoprefixer, cssnano и другие приятные и удобные штуковины. Буду продвигать то, что использую регулярно. Нам понадобится postcss-загрузчик. Также по мере необходимости установим autoprefixer.

npm install postcss-loader --save-dev

npm i -D autoprefixer

Спойлер: для рационального использования PostCSS вам не требуется Webpack, есть довольно приличный плагин post-css-cli, который позволяет вам использовать его в npm-скрипте.

Создайте postcss.config.js, где вам требуюся соответствующие плагины, вставьте:

module.exports = {

plugins: [

require('autoprefixer')

]

}

Теперь наш webpack.config.js должен выглядеть так:

// Webpack v4

const path = require('path');

// const ExtractTextPlugin = require('extract-text-webpack-plugin');

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

const WebpackMd5Hash = require('webpack-md5-hash');

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

module.exports = {

entry: { main: './src/index.js' },

output: {

path: path.resolve(\_\_dirname, 'dist'),

filename: '[name].[chunkhash].js'

},

module: {

rules: [

{

test: /\.js$/,

exclude: /node\_modules/,

use: {

loader: "babel-loader"

}

},

{

test: /\.scss$/,

use: [ 'style-loader', MiniCssExtractPlugin.loader, 'css-loader', 'postcss-loader', 'sass-loader']

}

]

},

plugins: [

new CleanWebpackPlugin('dist', {} ),

// new ExtractTextPlugin(

// {filename: 'style.[hash].css', disable: false, allChunks: true }

// ),

new MiniCssExtractPlugin({

filename: 'style.[contenthash].css',

}),

new HtmlWebpackPlugin({

inject: false,

hash: true,

template: './src/index.html',

filename: 'index.html'

}),

new WebpackMd5Hash()

]

};

Пожалуйста, обратите внимание на порядок плагинов, которые мы используем для .scss

use: [ 'style-loader', MiniCssExtractPlugin.loader, 'css-loader', 'postcss-loader', 'sass-loader']

Загрузчик инициализирует плагины, начиная с самого последнего.

Вы можете протестировать [autoprefixer](https://github.com/postcss/autoprefixer" \t "_blank), дополнив код в ваших scss-файлах и проверив вывод. Существует также способ исправить выход, указав, какой браузер вы хотите поддерживать в файле .browserslistrc.

Я бы посоветовала вам [PostCSS.parts](https://www.postcss.parts/" \t "_blank) для изучения плагинов, доступных для PostCSS, например:

* [utilities](https://github.com/ismamz/postcss-utilities);
* [cssnano](https://github.com/ben-eb/cssnano);
* [style-lint](https://github.com/stylelint/stylelint).

Я буду использовать cssnano для минимизации моего выходного файла и [css-mqpacker](https://github.com/hail2u/node-css-mqpacker" \t "_blank) для упорядочения медиа-запросов. Я также получила несколько подобных сообщений:

При желании можете попробовать cleancss.

Контроль версий

Ваш package.json на этом этапе должен иметь следующую структуру:

{

"name": "post",

"version": "1.0.0",

"description": "",

"main": "index.js",

"scripts": {

"build": "webpack — mode production",

"dev": "webpack — mode development"

},

"author": "",

"license": "ISC",

"devDependencies": {

"autoprefixer": "^8.2.0",

"babel-core": "^6.26.0",

"babel-loader": "^7.1.4",

"babel-preset-env": "^1.6.1",

"clean-webpack-plugin": "^0.1.19",

"css-loader": "^0.28.11",

"html-webpack-plugin": "^3.2.0",

"mini-css-extract-plugin": "^0.4.0",

"node-sass": "^4.8.3",

"postcss-loader": "^2.1.3",

"sass-loader": "^6.0.7",

"style-loader": "^0.20.3",

"webpack": "^4.4.1",

"webpack-cli": "^2.0.13",

"webpack-md5-hash": "0.0.6"

}

}

Очищаем ./dist

Мы можем попробовать импортировать clean-webpack-plugin, чтобы перед перегенерацией файлов очистить нашу папку ./dist.

// Webpack v4

const path = require('path');

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

const WebpackMd5Hash = require('webpack-md5-hash');

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

const CleanWebpackPlugin = require('clean-webpack-plugin');

module.exports = {

entry: { main: './src/index.js' },

output: {

path: path.resolve(\_\_dirname, 'dist'),

filename: '[name].[chunkhash].js'

},

module: {

rules: [

{

test: /\.js$/,

exclude: /node\_modules/,

use: {

loader: "babel-loader"

}

},

{

test: /\.scss$/,

use: [ 'style-loader', MiniCssExtractPlugin.loader, 'css-loader', 'postcss-loader', 'sass-loader']

}

]

},

plugins: [

new CleanWebpackPlugin('dist', {} ),

new MiniCssExtractPlugin({

filename: 'style.[contenthash].css',

}),

new HtmlWebpackPlugin({

inject: false,

hash: true,

template: './src/index.html',

filename: 'index.html'

}),

new WebpackMd5Hash()

]

};

Теперь у нас есть чистенькая и аккуратная конфигурация.