**Sass Loader**

Loads a Sass/SCSS file and compiles it to CSS.

Use the [css-loader](https://github.com/webpack-contrib/css-loader) or the [raw-loader](https://github.com/webpack-contrib/raw-loader) to turn it into a JS module and the [MiniCssExtractPlugin](https://github.com/webpack-contrib/mini-css-extract-plugin) to extract it into a separate file. Looking for the webpack 1 loader? Check out the [archive/webpack-1 branch](https://github.com/webpack-contrib/sass-loader/tree/archive/webpack-1).

**Install**

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

The sass-loader requires [node-sass](https://github.com/sass/node-sass) and [webpack](https://github.com/webpack) as [peerDependency](https://docs.npmjs.com/files/package.json" \l "peerdependencies). Thus you are able to control the versions accurately.

**Examples**

Chain the sass-loader with the [css-loader](https://github.com/webpack-contrib/css-loader) and the [style-loader](https://github.com/webpack-contrib/style-loader) to immediately apply all styles to the DOM.

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

// webpack.config.js

module.exports = {

...

module: {

rules: [{

test: /**\.**scss$/,

use: [

"style-loader", // creates style nodes from JS strings

"css-loader", // translates CSS into CommonJS

"sass-loader" // compiles Sass to CSS

]

}]

}

};

You can also pass options directly to [node-sass](https://github.com/andrew/node-sass) by specifying an options property like this:

// webpack.config.js

module.exports = {

...

module: {

rules: [{

test: /**\.**scss$/,

use: [{

loader: "style-loader"

}, {

loader: "css-loader"

}, {

loader: "sass-loader",

options: {

includePaths: ["absolute/path/a", "absolute/path/b"]

}

}]

}]

}

};

See [node-sass](https://github.com/andrew/node-sass) for all available Sass options.

**In production**

Usually, it's recommended to extract the style sheets into a dedicated file in production using the [MiniCssExtractPlugin](https://github.com/webpack-contrib/mini-css-extract-plugin). This way your styles are not dependent on JavaScript:

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

module.exports = {

...

module: {

rules: [{

test: /**\.**scss$/,

use: [

// fallback to style-loader in development

process.env.NODE\_ENV !== 'production' ? 'style-loader' : MiniCssExtractPlugin.loader,

"css-loader",

"sass-loader"

]

}]

},

plugins: [

new MiniCssExtractPlugin({

// Options similar to the same options in webpackOptions.output

// both options are optional

filename: "[name].css",

chunkFilename: "[id].css"

})

]

};

**Usage**

**Imports**

webpack provides an [advanced mechanism to resolve files](https://webpack.js.org/concepts/module-resolution/). The sass-loader uses node-sass' custom importer feature to pass all queries to the webpack resolving engine. Thus you can import your Sass modules from node\_modules. Just prepend them with a ~ to tell webpack that this is not a relative import:

@import "~bootstrap/dist/css/bootstrap";

It's important to only prepend it with ~, because ~/ resolves to the home directory. webpack needs to distinguish betweenbootstrap and ~bootstrap because CSS and Sass files have no special syntax for importing relative files. Writing @import "file" is the same as @import "./file";

**Problems with url(...)**

Since Sass/[libsass](https://github.com/sass/libsass) does not provide [url rewriting](https://github.com/sass/libsass/issues/532), all linked assets must be relative to the output.

* If you're just generating CSS without passing it to the css-loader, it must be relative to your web root.
* If you pass the generated CSS on to the css-loader, all urls must be relative to the entry-file (e.g. main.scss).

More likely you will be disrupted by this second issue. It is natural to expect relative references to be resolved against the .scssfile in which they are specified (like in regular .css files). Thankfully there are a two solutions to this problem:

* Add the missing url rewriting using the [resolve-url-loader](https://github.com/bholloway/resolve-url-loader). Place it before the sass-loader in the loader chain.
* Library authors usually provide a variable to modify the asset path. [bootstrap-sass](https://github.com/twbs/bootstrap-sass) for example has an $icon-font-path. Check out [this working bootstrap example](https://github.com/webpack-contrib/sass-loader/tree/master/test/bootstrapSass).

**Extracting style sheets**

Bundling CSS with webpack has some nice advantages like referencing images and fonts with hashed urls or [hot module replacement](https://webpack.js.org/concepts/hot-module-replacement/) in development. In production, on the other hand, it's not a good idea to apply your style sheets depending on JS execution. Rendering may be delayed or even a [FOUC](https://en.wikipedia.org/wiki/Flash_of_unstyled_content) might be visible. Thus it's often still better to have them as separate files in your final production build.

There are two possibilities to extract a style sheet from the bundle:

* [extract-loader](https://github.com/peerigon/extract-loader) (simpler, but specialized on the css-loader's output)
* [extract-text-webpack-plugin](https://github.com/webpack-contrib/extract-text-webpack-plugin) (more complex, but works in all use-cases)

**Source maps**

To enable CSS source maps, you'll need to pass the sourceMap option to the sass-loader *and* the css-loader. Yourwebpack.config.js should look like this:

module.exports = {

...

devtool: "source-map", // any "source-map"-like devtool is possible

module: {

rules: [{

test: /**\.**scss$/,

use: [{

loader: "style-loader"

}, {

loader: "css-loader", options: {

sourceMap: true

}

}, {

loader: "sass-loader", options: {

sourceMap: true

}

}]

}]

}

};

If you want to edit the original Sass files inside Chrome, [there's a good blog post](https://medium.com/@toolmantim/getting-started-with-css-sourcemaps-and-in-browser-sass-editing-b4daab987fb0). Checkout [test/sourceMap](https://github.com/webpack-contrib/sass-loader/tree/master/test) for a running example.

**Environment variables**

If you want to prepend Sass code before the actual entry file, you can set the data option. In this case, the sass-loader will not override the data option but just append the entry's content. This is especially useful when some of your Sass variables depend on the environment:

{

loader: "sass-loader",

options: {

data: "$env: " + process.env.NODE\_ENV + ";"

}

}

**Please note:** Since you're injecting code, this will break the source mappings in your entry file. Often there's a simpler solution than this, like multiple Sass entry files.

**mini-css-extract-plugin**

This plugin extract CSS into separate files. It creates a CSS file per JS file which contains CSS. It supports On-Demand-Loading of CSS and SourceMaps.

It builds on top of a new webpack v4 feature (module types) and requires webpack 4 to work.

Compared to the extract-text-webpack-plugin:

* Async loading
* No duplicate compilation (performance)
* Easier to use
* Specific to CSS

TODO:

* HMR support

**Install**

npm install --save-dev mini-css-extract-plugin

**Usage**

**Configuration**

**Minimal example**

**webpack.config.js**

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

module.exports = {

plugins: [

new MiniCssExtractPlugin({

// Options similar to the same options in webpackOptions.output

// both options are optional

filename: "[name].css",

chunkFilename: "[id].css"

})

],

module: {

rules: [

{

test: /**\.**css$/,

use: [

MiniCssExtractPlugin.loader,

"css-loader"

]

}

]

}

}

**Advanced configuration example**

This plugin should be used only on production builds without style-loader in the loaders chain, especially if you want to have HMR in development.

Here is an example to have both HMR in development and your styles extracted in a file for production builds.

(Loaders options left out for clarity, adapt accordingly to your needs.)

**webpack.config.js**

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

const devMode = process.env.NODE\_ENV !== 'production'

module.exports = {

plugins: [

new MiniCssExtractPlugin({

// Options similar to the same options in webpackOptions.output

// both options are optional

filename: devMode ? '[name].css' : '[name].[hash].css',

chunkFilename: devMode ? '[id].css' : '[id].[hash].css',

})

],

module: {

rules: [

{

test: /**\.**s?[ac]ss$/,

use: [

devMode ? 'style-loader' : MiniCssExtractPlugin.loader,

'css-loader',

'postcss-loader',

'sass-loader',

],

}

]

}

}

**Minimizing For Production**

While webpack 5 is likely to come with a CSS minimizer built-in, with webpack 4 you need to bring your own. To minify the output, use a plugin like [optimize-css-assets-webpack-plugin](https://github.com/NMFR/optimize-css-assets-webpack-plugin). Setting optimization.minimizer overrides the defaults provided by webpack, so make sure to also specify a JS minimizer:

**webpack.config.js**

const UglifyJsPlugin = require("uglifyjs-webpack-plugin");

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

const OptimizeCSSAssetsPlugin = require("optimize-css-assets-webpack-plugin");

module.exports = {

optimization: {

minimizer: [

new UglifyJsPlugin({

cache: true,

parallel: true,

sourceMap: true // set to true if you want JS source maps

}),

new OptimizeCSSAssetsPlugin({})

]

},

plugins: [

new MiniCssExtractPlugin({

filename: "[name].css",

chunkFilename: "[id].css"

})

],

module: {

rules: [

{

test: /**\.**css$/,

use: [

MiniCssExtractPlugin.loader,

"css-loader"

]

}

]

}

}

**Features**

**Using preloaded or inlined CSS**

The runtime code detects already added CSS via <link> or <style> tag. This can be useful when injecting CSS on server-side for Server-Side-Rendering. The href of the <link> tag has to match the URL that will be used for loading the CSS chunk. Thedata-href attribute can be used for <link> and <style> too. When inlining CSS data-href must be used.

**Extracting all CSS in a single file**

Similar to what [extract-text-webpack-plugin](https://github.com/webpack-contrib/extract-text-webpack-plugin) does, the CSS can be extracted in one CSS file usingoptimization.splitChunks.cacheGroups.

**webpack.config.js**

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

module.exports = {

optimization: {

splitChunks: {

cacheGroups: {

styles: {

name: 'styles',

test: /**\.**css$/,

chunks: 'all',

enforce: true

}

}

}

},

plugins: [

new MiniCssExtractPlugin({

filename: "[name].css",

})

],

module: {

rules: [

{

test: /**\.**css$/,

use: [

MiniCssExtractPlugin.loader,

"css-loader"

]

}

]

}

}

**Extracting CSS based on entry**

You may also extract the CSS based on the webpack entry name. This is especially useful if you import routes dynamically but want to keep your CSS bundled according to entry. This also prevents the CSS duplication issue one had with the ExtractTextPlugin.

const path = require('path');

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

function recursiveIssuer(m) {

if (m.issuer) {

return recursiveIssuer(m.issuer);

} else if (m.name) {

return m.name;

} else {

return false;

}

}

module.exports = {

entry: {

foo: path.resolve(\_\_dirname, 'src/foo'),

bar: path.resolve(\_\_dirname, 'src/bar')

},

optimization: {

splitChunks: {

cacheGroups: {

fooStyles: {

name: 'foo',

test: (m,c,entry = 'foo') => m.constructor.name === 'CssModule' && recursiveIssuer(m) === entry,

chunks: 'all',

enforce: true

},

barStyles: {

name: 'bar',

test: (m,c,entry = 'bar') => m.constructor.name === 'CssModule' && recursiveIssuer(m) === entry,

chunks: 'all',

enforce: true

}

}

}

},

plugins: [

new MiniCssExtractPlugin({

filename: "[name].css",

})

],

module: {

rules: [

{

test: /**\.**css$/,

use: [

MiniCssExtractPlugin.loader,

"css-loader"

]

}

]

}

}