


MiniCssExtractPlugin

npm v0.8.0

dependencies up to date

 Azure Pipelines never built

 codecov 87%

gitter webpack/webpack

This plugin extracts 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

Install

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

Usage

Configuration

`publicPath`

Type: `String|Function` Default: the `publicPath` in `webpackOptions.output`

Specifies a custom public path for the target file(s).

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
      // all options are optional
      filename: '[name].css',
      chunkFilename: '[id].css',
      ignoreOrder: false, // Enable to remove warnings about conflicting order
    }),
  ],
  module: {
    rules: [
      {
        test: /\.css$/,
        use: [
          {
            loader: MiniCssExtractPlugin.loader,
            options: {
              // you can specify a publicPath here
              // by default it uses publicPath in webpackOptions.output
              publicPath: '../',
              hmr: process.env.NODE_ENV === 'development',
            },
          },
          'css-loader',
        ],
      },
    ],
  },
};
```

publicPath function 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: [
        {
          loader: MiniCssExtractPlugin.loader,
          options: {

            publicPath: (resourcePath, context) => {
              // publicPath is the relative path of the resource to the context
              // e.g. for ./css/admin/main.css the publicPath will be ../../
              // while for ./css/main.css the publicPath will be ../
              return path.relative(path.dirname(resourcePath), context) + '/';
            },
          },
        },
        '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: /\.sa|sc|css$/,
      use: [
        {
          loader: MiniCssExtractPlugin.loader,
          options: {
            hmr: process.env.NODE_ENV === 'development',
          },
        },
        'css-loader',
        'postcss-loader',
        'sass-loader',
      ],
    },
  ],
},
];
};

```

Hot Module Reloading (HMR)

extract-mini-css-plugin supports hot reloading of actual css files in development. Some options are provided to enable HMR of both standard stylesheets and locally scoped CSS or CSS modules. Below is an example configuration of mini-css for HMR use with CSS modules.

While we attempt to hmr css-modules. It is not easy to perform when code-splitting with custom chunk names. `reloadAll` is an option that should only be enabled if HMR isn't working correctly. The core challenge with css-modules is that when code-split, the chunk ids can and do end up different compared to the filename.

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: [
      {
        loader: MiniCssExtractPlugin.loader,
        options: {
          // only enable hot in development
          hmr: process.env.NODE_ENV === 'development',
          // if hmr does not work, this is a forceful method.
          reloadAll: true,
        },
      },
    ],
    'css-loader',
  ],
},
],
},
};

```

Minimizing For Production

To minify the output, use a plugin like [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 TerserJSPlugin = require('terser-webpack-plugin');
const MiniCssExtractPlugin = require('mini-css-extract-plugin');
const OptimizeCSSAssetsPlugin = require('optimize-css-assets-webpack-plugin');
module.exports = {
  optimization: {
    minimizer: [new TerserJSPlugin({}), 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. The `data-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](#) does, the CSS can be extracted in one CSS file using `optimization.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'],
  },
],
},
};
```

Module Filename Option

With the `moduleFilename` option you can use chunk data to customize the filename. This is particularly useful when dealing with multiple entry points and wanting to get more control out of the filename for a given entry point/chunk. In the example below, we'll use `moduleFilename` to output the generated css into a different directory.

```
const miniCssExtractPlugin = new MiniCssExtractPlugin({
  moduleFilename: ({ name }) => `${name.replace('/js/', '/css/')}.css`,
});
```

Long Term Caching

For long term caching use `filename: "[contenthash].css"`. Optionally add `[name]`.

Remove Order Warnings

For projects where css ordering has been mitigated through consistent use of scoping or naming conventions, the css order warnings can be disabled by setting the `ignoreOrder` flag to true for the plugin.

```
new MiniCssExtractPlugin({
  ignoreOrder: true,
}),
```

Media Query Plugin

If you'd like to extract the media queries from the extracted CSS (so mobile users don't need to load desktop or tablet specific CSS anymore) you should use one of the following plugins:

- [Media Query Plugin](#)
- [Media Query Splitting Plugin](#)

License

MIT