Tobias Koppers
May 28, 2017 · 6 min read

The new CSS workflow (step 1)

Current state, minimal css-loader, ICSS2

> *Note: This is a technical document with many internal details. You have to be deeply involved with webpack to understand all of it.*

Current state

We assume simple CSS workflow:

- which uses `cssnext` to write future CSS
- minimize CSS in production
- imports `normalize.css` for normalizing CSS

In the current state this can be achieved with a config like this:

```
rules: [
  {
    test: /\.css$/,
    use: [
      "style-loader",
      { loader: "css-loader", options: {
importLoaders: 1 },
      { loader: "postcss-loader", options: {
        plugins: { "cssnext": {} }
      }}
    ]
  }
]
```

In our example we use a CSS file like this one:

```
body {
  background: url(image.png);
  overflow-wrap: break-word;
}
```

```
var css = require("-!css-loader??ref-0-1!postcss-
loader??ref-0-2!file.css");
require("./addStyle")(css);
```

The `require` expression in the first line creates a new module, which first loads the css file:

```
body {
  background: url(image.png);
  overflow-wrap: break-word;
}
```

Then the file is processed with the postcss-loader giving the following result:

```
body {
  background: url(image.png);
  word-wrap: break-word;
}
```

This result is passed into the css-loader, which generates a

```
module.exports = [
  [
    module.id,
    "body {\n  background: url(\"" +
      require("./image.png") +
    "\");\n  word-wrap: break-word;\n}"
  ]
];
```

Why include `module.id`? The style-loader deduplicates styles according to this id.

• • •

This workflow embeds the CSS into the JS bundle, but technically it **can** be better to use separate CSS files. This causes an extra request (2 files for a chunk), but CSS can be processed in parallel by the browser and is cached separately. For SSR, it is important that the CSS can be loaded before the JS.

The current workflow for separate CSS files involves the extract-text-webpack-plugin, which is a big hack that executes the result of the css-loader and creates a CSS file contains only the current module targeting node.js and evaluate it. You can guess it: This isn't the most performant solution.

The big plan

In the long term we want to make it possible to add first-class module support for CSS to webpack. This will work the following way:

- We add a new module type to webpack: Stylesheet (next to Javascript)
- We adjust the Chunk Templates to write two files. One for the javascript and one of the stylesheets (in a `.css` file).
- We adjust the chunk loading logic to allow loading of stylesheets. We need to wait for CSS applied or at least loaded, before executing the JS.
- When we generate a chunk load we may load the js chunk and the stylesheet chunk in parallel (combined by `Promise.all`).

> *Hidden gem: Replace Stylesheet with WebAssembly and `.css` with `.wasm` to get the plan for WebAssembly. They both fit together really well…*

This has a few benefits:

- We can generate stylesheet files for on-demand-chunks (this was not possible with the extract-text-webpack-plugin)
- Using stylesheets is a lot easier compared to the extract-text-webpack-plugin
- Separate stylesheets will be the default workflow
- Stylesheets can be cached independent for javascript
- Stylesheet is only parsed once (by the css parser) compared to style-loader (by the js parser as string + the css parser)

But also a few limitations:

- On demand loading with stylesheets will cause two requests instead of one compared to style-loader.

- All stylesheet content must be statically discoverable. You can't generate stylesheet content at runtime.

- `publicPath` for stylesheet or referenced assets can't be provided at runtime. (This is also a limitation of the extract-text-webpack-plugin.)

The first-class CSS support will replace `style-loader` and `css-loader`:

```
rules: [
  {
    test: /\.css$/,
    type: "stylesheet", // probably also the
default for .css
    use: [
      { loader: "postcss-loader", options: {
        plugins: { "cssnext": {} }
      }}
    ]
  }
]
```

`style-loader` and `css-loader` won't be deleted, so you could still fallback to them if you want embedded

The steps

Step 1: Cleanup css-loader

Currently the css-loader does a lot stuff:

- Handle `@import`
- Handle ICSS `:import` and `:export`
- Handle `url()`
- Handle `~module`
- Minimizing
- SourceMapping
- Aliasing
- Forking nested loaders (`importLoaders`)

This is a lot and also causes performance issues in the current state. In the first step we want to reduce the feature set and try to create the minimal possible `css-loader`. This creates only a small API surface for CSS. Keep in mind that the same API will be used for first-class

So what is the minimal feature set for the css-loader? Very similar to JS modules, CSS module has code, dependencies and exports. For JS we have CommonJS, AMD and ES Modules. For CSS there is no official standard, but for the CSS Modules Initiative Glen Maddern developed one: Interoperable CSS. At the time of writing ESM was not a thing yet, so it need to be adapted to ESM, but that only a minor change.

With ICSS support the minimal feature set for the css-loader would be ICSS + `@import`. I discussed this with the webpack contrib team and they kept asking: Why do we remove `url()`, but keep `@import`? “For technical reasons” is probably a bad answer when designing an API. So we changing ICSS anyway (ESM), so why not adding the missing parts to replace `@import` with ICSS.

So ICSS2 is ICSS plus the following changes: ESM, media queries for `:import` and CSS rules can be imported with `:import`.

ICSS2

Here a short explanation of Interoperable CSS v2:

```
:import(<request>) {
  import: <exportName> <media queries>;
  <alias>: <exportName>;
  <alias>: <exportName>;
}
```

The `:import` rule allows to declare a dependency of the file. `<request>` points to the imported module. The request is resolved according to the normal resolving rules (`./relative` `../relative` `module` `module/path`). The special `import` key allows to import the CSS rules from an export of the imported module. Optionally this can be imported conditionally with media queries (`@import` also allows media queries, we don’t want to lose this behavior). Any other key is treated as alias. Every occurrence (as identifier) of this alias is replaced with the value of the export of the imported module.

```
:export {
  <exportName>: <any value>;
  <exportName>: <any value>;
}
```

`:export` is pretty simple. The key is the exported name (must be a valid JS identifier, ESM). The string passed as value is exported. Spacing is insignificant and the spacing of the value is reduced to a single space per gap, without leading and training spaces.

The CSS content is exported as `default` export. The format is implementation-specific, but the implementation must ensure that files imported multiple times only occur once in the result and that the order of the imported CSS is kept (if possible).

css-loader with ICSS2 example

```
@import "mobile.css" (max-width: 400px);
body {
  background: url(./image.png);
}
```

With new PostCSS plugins (or plugins for your preprocessor) it will be transformed into this piece of ICSS2:

```
:import("mobile.css") {
  import: default (max-width: 400px);
}
:import("./image.png") {
  __url_image_png: default;
}
body {
  background: url(__url_image_png);
}
```

So the CSS loader resp. the CSS support in webpack only needs two keywords to do it’s job: `:import` and `:export`. Super simple. Very performant, because parsing this doesn’t need a full AST, a string-only transformation will do it.

FAQ

> *So we are no longer able to use CSS Modules with the css-loader?*

Yes and no. The css-loader will no longer support it, but with ICSS2 it still support the building blocks for CSS Modules. There will be a separate loader or postcss plugin for CSS Modules transforming the CSSM syntax into

> *Do I still need to use `~module` to reference from modules?*

This behavior will no longer be handled by the css-loader. It will be part of the postcss plugins for `@import` resp. `url()`. There will be an option selecting between standard requests and modular requests.

> *Since `importLoaders` will be removed, how can I specify loaders used on imported resources?*

The css-loader will no longer override loaders on imported resource. webpack will be responsible for determining loaders. This means `module.rules` apply to imported loaders too. Currently this requires a more complex configuration to apply the `style-loader` only on CSS files imported from JS:

```
rules: [
  {
    test: /\.css$/,
    rules: [
      {
        issuer: { not: /\.css$/ }
      {
        use: "css-loader"
      }
    ]
  }
]
```

But this now allows you to mix different compile-to-CSS languages. Yeah!

• • •

With this first step we laid the base for a performant css-loader and a minimal implementation of CSS in webpack.

Stay tuned for more info about the next steps: How we plan to integrate CSS as first-class citizen into webpack.

• • •

webpack is not backed by a big company, unlike many other big Open Source products. The development is funded by donations. *Please consider donating if you depend on webpack… (Ask your boss!)*

- Capital One (Bank) donated a total of $12,000
- ag-Grid (DataGrid) donated a total of $7,500
- AngularClass (Training) donated a total of $2,512
- Google Angular (Framework) donated a total of $2,500
- OpenCollective (OS funding) donated a total of $1,490
- Full list

JavaScript

Tobias Koppers