CDS 2020: Moving to Modern Javascript

As far as Handling both current and older browsers has gotten more complicated since Javascript adopted an anual realease schedule.

We are not talking about the stark difference between ES2015/ES6 and ES5 but on a more nuanced set of difference and what it means for transpilation and browser support.

Take for example, Async/Await, a features introduced in ES2017. If we want to use it in production we have have to provide fallback for browsers that may meet all our other Javascript requirements. Take the following list of Javascript features along with the version of the langaguage they were introduced in:

- Classes (ES2015)
- Arrow functions (ES2015)
- Generators (ES2015)
- Block scoping (ES2015)
- Destructuring (ES2015)
- Rest parameters and spread syntax (ES2015)
- Object shorthand (ES2015)
- Async/await (ES2017)

If we want to support all the items in the list, then we have to support ES2017 for `async/await` and assume that if the browsers supports ES2017 then it'll support ES2015 features ask well.

With the information above, and for the purpose of this post, we'll stick with ES2017 as the closest to modern syntax we have today.

We can use the following browserslist configuration as a starting point. We're basically telling browserslist-aware tools that to execute the assigned task for all browsers older than these versions.

Normally I would have used a yearly preset like preset-es2017 but the Babel team now discourages using those and are deprecated since Babel 6.

```
  "browserslist": [
    "Firefox < 84",
"Firefox < 84"87",
    "Edge < 87",
    "Safari < 13",
    "iOS < 13.7"
  ],
```

Babel provides two plugins that use these lists are @babel/preset-env and @babel/preset-modules to make your WebPack bundles leaner by only compiling code for browsers on your list. `preset-modules` does some additional optimizations for modules that are not part of `preset-env` yet.

Also see Philip Walton's Deploying ES2015+ Code in Production Today for examples of WebPack configurations to build both modern and legacy code bundles

Rollup has its own rollup-plugin-babel to handle processing with Babel.

Serving content to different browsers

The best way to serve different files based on javascript support is to use the module/nomodule pattern.

Using `type="module"` on script tags for modern files allows you to use modules and modern features on your browser code today.

For browsers that don't support the features we want `nomodule` attribute for the legacy files.

```
<script type="module" src="modern.js"></script>
<script nomodule src="legacy.js"></script>
```

Browsers that support type="module" will load `modern.js`, and leagacy browsers will ignore it.

Package exports in Node

In our package.json file, we can now define another “main” script file using the exports key.

```
{
  "name": "my-packag"my-package""exports"odern.js",
  "main": "./l": ": "./legacy.js"
}
```

The idea is that if a version of Node can read the exports filed in the `package.json` file, then it’ll execute that script and if it doesn’t then it’ll execute the script in the main entry point.

This exports field is only supported in Node 12.8 and above, which implies support for ES2019+ syntax.