Baseline for Javascript development: modules in the browser without transpilation

Last April, I wrote Evergreen browsers and looking for a sane JS baseline looking for a sane Javaascript baseline where I could write scripts without having to worry about transpilation for those browsers that didn't support the features.

I took a basic set of features from Publish, ship, and install modern JavaScript for faster applications) and wrote about how to use them in ESModules that would run in Node.js.

This post follows up on that one and will cover two approaches to using ESModules in the browser.

The process below assumes that you already have a browserslist file or section in your package.json file ready to use. See the Browserslist Readme for more information about configuring your browserslist file.

To get this baseline level of module support, we can use babel and different plugins depending to achieve our different levels of support:

- @babel/preset-env to transpile out code for older browsers down to ES5 using a list of target browsers using a browserslist compatible file
    - It leverages multiple data sources to maintain mappings of which version of the supported target environments gained support of a JavaScript syntax or browser feature, as well as a mapping of those syntaxes and features to Babel transform plugins and core-js polyfills
- @babel/preset-modules plugin to smooth out the features we want to use in browsers that support some of our target features but not others
    - Here's the problem that `preset-modules` addresses: if any version of any browser in that the target list contains a bug triggered by modern syntax, the only solution is to enable the corresponding transform group that fixes that bug. In other words, `preset-env` converts code to ES5 in order to get around syntax bugs in ES2017

- `preset-modules` compiles the problem code to the closest functioning syntax that is supported by the target browsers

We can make both `preset-env` and `preset-modules` work together in the same Babel configuration.

First step is to install the plugins:

```
npm install -D @babel/preset-env @babel/preset-modules
```

Next, we enable the plugins in our Babel configuration:

```
{
  "presets": [
    "@babel/prese"@babel/preset-env"/preset-modules"
  ]
}
```

Lastly, configure two environments to generate the different bundles of our code, this is one possible way to do it:

```
{
  "env": {
    "modern": {
      "presets": [
        "@babel/preset-modules"
      ]
    },
    "@babel/preset-modules""legacy"        "@babel/preset-env"
      ]
    }
  }
}
"@babel/preset-env"
```

Once we create the bundles, we can use them together in the module/nomodule pattern.

```
<!-- transpiled with preset-modules: -->
<script
  type="module"
  src="modern.js<script
  type="module"
  src="modern.js">
<!-- transpiled with preset-env: -->acy.js">
</script>
</script>
</script>
```

The modern.js branch will work in browsers that support modules and will use code that has been optimised for those browsers.

If a browser doesn't support modules (unlikely but possible) then it'll run the legacy.js script and will use code transpiled to ES5.