Evergreen browsers and looking for a sane JS baseline

> The term “evergreen browser” refers to browsers that are automatically upgraded to future versions, rather than being updated by distribution of new versions from the manufacturer, as was the case with older browsers.
>
> from Techopedia — Evergreen Browser

The concept of an evergreen browser is an interesting one from a development perspective but it’s not without its drawbacks.

In an evergreen world developers don’t have to worry about what version of a browser a user has because it automatically updates to the latest version. Most major browsers are evergreen, either via autoupdating (Chromium browsers and Firefox) or via OS updates (like Safari).

At least that’s the theory.

I can think of at least two reasons why people would stop auto updating to the latest version of a given browser.

There may be reasons why corporate IT departments want to stop browser updates, likely because there’s an issue preventing internal applications from working or the IT team needs to test an approve updates before updating hundreds or potentially thousands of computers.

Other users may stop auto-updating their browsers on the mistaken notion that they will be able to keep Adobe Flash working past its end of life and despite the blocking of Flash content on the latest released version of the Flash Player plugin before its termination.

So, even for a short time there may be users who can’t use apps that rely on the latest features.

Think features, not specific versions

So what do we do? We want to leverage modern features but we don't want to transpile our code if we don't have to but at the same time we want to support as large a browser base as possible.

The first thing is not to worry about specific versions of the ECMAScript specification, but rather in syntax that is supported by your target browsers.

As long as modern browsers (Chrome, Edge, Firefox, and Safari) support a given feature, we're safe. These browsers make up more than 90% of the market, with other browsers that rely on the same rendering engines make up an additional 5%. Let's look at some interesting features from older specifications that are available across browsers without needing to polyfill (next to the feature is the version of the ECMAscript specification where it was introduced)

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

So, if we choose to use these features as the core of our JS code, we don't need to transpile them to support older browsers.

Features in newer versions of the language specification generally have less consistent support across modern browsers, not enough to rely on those features in production environments.

Using modules in Node

One of the things that has really bothered me when working with Node is that getting modules and imports to work on Node-based projects.

The first approach is to pay attention to the extension you use:

Node treats following items as ES modules when passed to node as the initial

input, or when referenced by import:

- Files ending in `.mjs`
- Files ending in `.js`, or extensionless files, when the nearest parent package.json file with a `type="module"` field

Node.js will treat as CommonJS all other forms of input without a `type` field in `package.json`, or string input without the flag `--input-type`; this is meant to provide backward compatibility. However, now that Node supports both CommonJS and ES modules, we should always be explicit with the type of modules we are working with.

Node.js will treat the following as CommonJS when passed to node as the initial input, or when referenced by import:

- Files ending in `.cjs` Files ending in `.js`, or extensionless files, when the nearest parent package.json file contains a top-level field “type” with a value of “commonjs”

The second approach is to add fields to the `package.json` file. Node.js has standardized an “exports” field to define entry points for a package:

```
{
  "exports": "./index.js"
}
```

Modules referenced by the “exports” field imply Node 12.8 or later, which supports ES2019. This means that any module referenced using the “exports” field can be written in modern JavaScript.

When consuming a module with an “exports” field we must assume that it contains modern code and transpile if necessary.

However, if you choose not to transpile the code in a package with only “exports” field, it won’t be usable unless the browser supports the modern code you wrote.

To make sure that the code will work with out transpilation use both the “exports” and “main” fields in your package.json file.

“exports” will provide the modern code version and “main” will point to a ES5

+ CommonJS version usable in older browsers.

```
{
  "name": "foo",
  "foo""exports": "./modern.js",
  "main": "./legacy.cjs"
}
```

There is one more thing we can do to optimize our code for both modern and legacy browsers.

We can define a “module” field pointing to a second legacy bundle that uses JavaScript module syntax (import and export).

```
{
  "name": "foo",
  "foo""exports": "./modern.js",
  "main": "./legacy.cjs",
  "module": "./module.js"
}
```

Bundlers, like Webpack and Rollup, use the “module” field to take advantage of module features and enable tree shaking.

This is a legacy bundle that does not contain any modern code other than import/export statements. Use this method to ship a legacy fallback that is optimized for bundling.

See Publish, ship, and install modern JavaScript for faster applications for more information and more tools you can use when working with modern versus legacy code and Announcing core Node.js support for ECMAScript modules for information about Node module support.