

# An overview of Javascript tooling

This article was originally published at

<https://gist.github.com/joepie91/3381ce7f92dec7a1e622538980c0c43d>.

Getting confused about the piles of development tools that people use for Javascript? Here's a quick index of what is used for what.

**Keep in mind that you shouldn't add tools to your workflow for the sake of it.** While you'll see many production systems using a wide range of tools, these tools are typically used because they solved a *concrete problem* for the developers working on it. You should **not** add tools to your project unless you have a concrete problem that they can solve; none of the tools here are *required*.

Start with nothing, and add tools as needed. This will keep you from getting lost in an incomprehensible pile of tooling.

## Build/task runners

**Typical examples:** Gulp, Grunt

These are not exactly build tools in and of themselves; they're rather just used to glue together *other* tools. For example, if you have a set of build steps where you need to run tool A after tool B, a build runner can help to orchestrate those tools.

## Bundlers

**Typical examples:** Browserify, Webpack, Parcel

These tools take a bunch of `.js` files that use [modules](#) (either CommonJS using `require()` statements, or ES Modules using `import` statements), and combine them into a *single* `.js` file. Some of them also allow specifying 'transformation steps', but their main purpose is bundling.

Why does bundling matter? While in Node.js you have access to a module system that lets you load files as-needed from disk, this wouldn't be practical in a browser; fetching every file individually

over the network would be very slow. That's why people use a bundler, which effectively does all this work upfront, and then produces a single 'combined' file with all the same guarantees of a module system, but that can be used in a browser.

Bundlers can also be useful for running module-using code in very basic JS environments that don't have module support for some reason; this includes Google Sheets, extensions for PostgreSQL, GNOME, and so on.

**Bundlers are *not* transpilers.** They do not compile one language to another, and they don't "make ES6 work everywhere". Those are the job of a *transpiler*. Bundlers are sometimes configured to *use* a transpiler, but the transpiling itself isn't done by the bundler.

**Bundlers are *not* task runners.** This is an especially popular misconception around Webpack. Webpack does *not* replace task runners like Gulp; while Gulp is designed to glue together arbitrary build tasks, Webpack is specifically designed for *browser bundles*. It's commonly useful to use Webpack *with* Gulp or another task runner.

## Transpilers

**Typical examples:** Babel, the TypeScript compiler, CoffeeScript

These tools take a bunch of code in one language, and 'compile' it to another language. They're called commonly 'transpilers' rather than 'compilers' because unlike traditional compilers, these tools don't compile to a lower-level representation; they're just different languages at a similar level of abstraction.

These are typically used to run code written against newer JS versions in older JS runtimes (eg. Babel), or to provide custom languages with more conveniences or constraints that can then be executed in any regular JS environment (TypeScript, CoffeeScript).

## Process restarters

**Typical examples:** nodemon

These tools automatically restart your (Node.js) process when the underlying code is changed. This is used for development purposes, to remove the need to manually restart your process every change.

A process restarter may either watch for file changes itself, or be controlled by an external tool like a build runner.

## Page reloaders

**Typical examples:** LiveReload, BrowserSync, Webpack hot-reload

These tools automatically refresh a page in the browser and/or reload stylesheets and/or re-render parts of the page, to reflect the changes in your *browser-side* code. They're kind of the equivalent of a process restarter, but for webpages.

These tools are usually externally controlled; typically by either a build runner or a bundler, or both.

# Debuggers

**Typical examples:** Chrome Developer Tools, node-inspect

These tools allow you to inspect *running* code; in Node.js, in your browser, or both. Typically they'll support things like pausing execution, stepping through function calls manually, inspecting variables, profiling memory allocations and CPU usage, viewing execution logs, and so on.

They're typically used to find tricky bugs. It's a good idea to learn how these tools work, but often it'll still be easier to find a bug by just 'dumb logging' variables throughout your code using eg.

`console.log`.

---

Revision #1

Created 11 December 2024 01:26:39 by joepie91

Updated 11 December 2024 18:17:59 by joepie91