Why webpack

To understand why you should use webpack, let's recap how we used JavaScript on the web before bundlers were a thing.

There are two ways to run JavaScript in a browser. First, include a script for each functionality; this solution is hard to scale because loading too many scripts can cause a network bottleneck. The second option is to use a big `.js` file containing all your project code, but this leads to problems in scope, size, readability and maintainability.

IIFE's - Immediately invoked function expressions

IIFEs solve scoping issues for large projects; when script files are wrapped by an IIFE, you can safely concatenate or safely combine files without worrying about scope collision.

The use of IIFEs led to tools like Make, Gulp, Grunt, Broccoli or Brunch. These tools are known as task runners, and they concatenate all your project files together.

However, changing one file means you have to rebuild the whole thing. Concatenating makes it easy to reuse scripts across files but makes build optimizations more difficult. How can you find out if code is actually being used or not?

Even if you only use a single function from lodash, you have to add the entire library and then squish it together. How do you treeshake the dependencies on your code? Lazy loading chunks of code can be hard to do at scale and requires a lot of manual work from the developer.

Birth of JavaScript Modules happened thanks to Node.js

webpack runs on Node.js, a JavaScript runtime that can be used in computers and

servers outside a browser environment.

When Node.js was released a new era started, and it came with new challenges. Now that JavaScript is not running in a browser, how are Node applications supposed to load new chunks of code? There are no html files and script tags that can be added to it.

CommonJS came out and introduced `require`, which allows you to load and use a module in the current file. This solved scope issues out of the box by importing each module as it was needed.

npm + Node.js + modules -- mass distribution

JavaScript is taking over the world as a language, as a platform and as a way to rapidly develop and create fast applications.

But there is no browser support for CommonJS. There are no live bindings. There are problems with circular references. Synchronous module resolution and loading is slow. While CommonJS was a great solution for Node.js projects, browsers didn't support modules, so bundlers and tools like Browserify, RequireJS and SystemJS were created, allowing us to write CommonJS modules that run in a browser.

ESM - ECMAScript Modules

The good news for web projects is that modules are becoming an official feature in the ECMAScript standard. However, browser support is incomplete and bundling is still faster and currently recommended over these early module implementations.

Automatic Dependency Collection

Old school Task Runners and even Google Closure Compiler requires you to manually declare all dependencies upfront. While bundlers like webpack automatically builds and infer your dependency graph based on what is imported and exported. This along with other plugins and loaders make for a great developer experience.

Wouldn't it be nice...

...to have something that will not only let us write modules but also support any

module format (at least until we get to ESM) and handle resources and assets at the same time?

This is why webpack exists. It's a tool that lets you bundle your JavaScript applications (supporting both ESM and CommonJS), and it can be extended to support many different assets such as images, fonts and stylesheets.

webpack cares about performance and load times; it's always improving or adding new features, such as async chunk loading and prefetching, to deliver the best possible experience for your project and your users.