Course: <https://frontendmasters.com/courses/webpack-fundamentals>

**Tóm tắt:**

**I.** Why webpack

+ Nêu vấn đề chèn script vào browser theo cách truyền thống thì hạn chế như thế nào? Chỉ ra giải pháp là sử dụng IIFE, giúp “safety” combines file, hạn chế vấn đề về scope. Nhược điểm: slow, dư thừa code, không hỗ trợ lazy load

+ Nêu ra các vấn đề CommonJS in node: không hỗ trợ browser, chậm, không hỗ trợ lazy load

+ Nêu các vấn đề về ESM trên browser: không hỗ trợ node, chậm, không hỗ trợ lazyload

+ Giới thiệu về webpack: hỗ trợ code splitting, lazy load, hỗ trợ nhiều module format (commonJS, ESM), loại bỏ code dư thưa.

II. Webpack from scratch

+ Cách cấu hình package.json environment dev, prod, debug với webpack

+ Giới thiệu các cách sử dụng module

+ Giải thích cách webpack bundle hoạt động

III. Webpack Core Concepts

+ Giải thích các khái niệm như entry point, output, loaders, plugins

+ Cách cấu hình file webpack config theo environment tương ứng

IV. Using Plugins

+ Tiến hành cài đặt các plugin quan trọng

+ Cấu hình preset, các cấu hình webpack mở rộng, không ảnh hưởng đến cấu hình environment

Contents

[I. Why Webpack 2](#_Toc48602057)

[1. Problems with Script Loading 2](#_Toc48602058)

[2. History of modules 4](#_Toc48602059)

[3. EcmaScript Modules (ESM) 5](#_Toc48602060)

[4. Introducing Webpack 6](#_Toc48602061)

[5. Configuring Webpack 6](#_Toc48602062)

[II. Webpack from scratch 6](#_Toc48602063)

[1. Adding npm Scripts for Environment Builds 6](#_Toc48602064)

[2. Setting Up Debugging 6](#_Toc48602065)

[3. Coding Your First Module 7](#_Toc48602066)

[4. CommonJS Export 8](#_Toc48602067)

[5. CommonJS Named Exports 9](#_Toc48602068)

[6. Tree Skaking 9](#_Toc48602069)

[7. Webpack Bundle Walkthrough 10](#_Toc48602070)

[III. Webpack Core Concepts 13](#_Toc48602071)

[1. Webpack Entry 13](#_Toc48602072)

[2. Output & Loaders 14](#_Toc48602073)

[3. Chaining Loaders 15](#_Toc48602074)

[4. Webpack Plugins 16](#_Toc48602075)

[5. Passing Variable to Webpack Config 17](#_Toc48602076)

[6. Adding Webpack Plugins 17](#_Toc48602077)

[7. Setting Up a Local Development Server 18](#_Toc48602078)

[8. Splitting Environment Config Files 19](#_Toc48602079)

[9. Webpack Q&A 19](#_Toc48602080)

[IV. Using Plugins 20](#_Toc48602081)

[1. Using CSS with Webpack 20](#_Toc48602082)

[2. Hot Module Replacement with CSS 20](#_Toc48602083)

[3. File Loader & URL Loader 21](#_Toc48602084)

[4. Loading Images with JavaScript 22](#_Toc48602085)

[5. Limit Filesize Option in URL Loader 23](#_Toc48602086)

[6. Implementing Presets 23](#_Toc48602087)

[7. Bundle Analyzer Preset 24](#_Toc48602088)

[8. Compression Plugin 25](#_Toc48602089)

[9. Source Maps 25](#_Toc48602090)

[V. Wrapping Up 27](#_Toc48602091)

1. Why Webpack
2. Problems with Script Loading

Why webpack? What is solving?

Only 2 ways to use JavaScript in the browser.

The first way is adding a script tag for passing a source attribute and a reference to that JS file

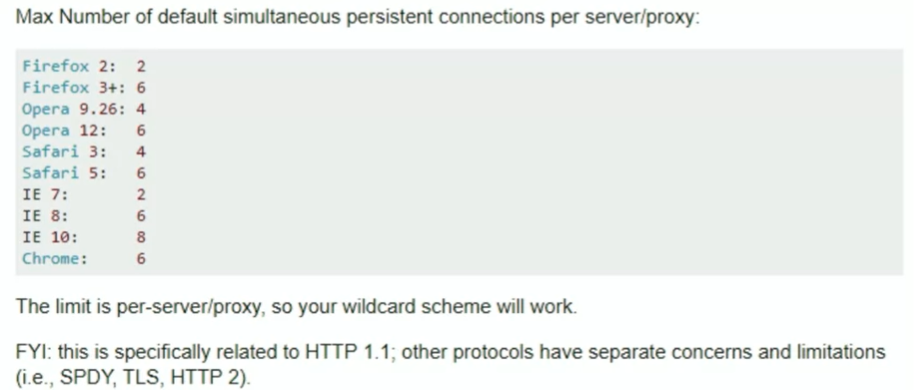


The second way is write your job description to html



* Problems:

+ Doen’t scale: to many script load from script tags in HTML



+ Unmaintainable scripts: everything is global in its scope.There’s a lot of issues with just scope and variable conflicts.

* Solutions?

+ IIFE:



* Treat each file as IIFE (revealing module). Concatenate => we can “safely” combine files without concern of scope collision

However, there’s still problems with this. Anytime you want to change one file, you have to rebuild the whole thing. If you’re just concatenating files together, how do you remove code that you’ve actually not using? Like, you need to adding the entire library for 1 function in Lodash.

IIFE is kinda slow. it forces JavaScript engines to eager parse the code, and so you incur much more intensive or costly parsing time when your JavaScript's loading, so it leads to slower loading apps.

And how do you lazy load with a task runner? You don't, well there is no way, whatever you're building.

1. History of modules

Node is JavaScript that runs on the server, how do you load JS if there’s no DOM?

How do you add a script tag if there is no HTML in node.

-> CommonJS have a syntax called required, which allows you to inject other pieces of a module into the current module



Problems:

+ There is no browser support for CommonJS

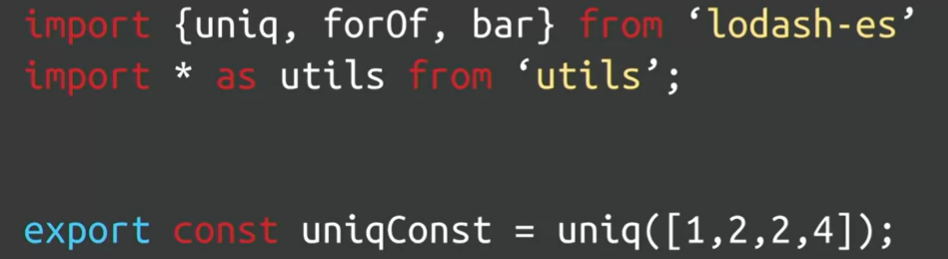
+ No live bindings (problems with circular references)

(It means that when the exporting module changes a value, the change will be visible from the importer side. This is not the case for CommonJS modules. Module exports are copied in CommonJS. Hence importing modules cannot see changes happened on the exporter side.)

+ the resolution algorithm for CommonJS is kinda slow, it's really slow (it's synchronous)

+ No static async/lazy loading (all bundles up front). Require is kind of dynamic, but there’s no way to asynchronously load st.

1. EcmaScript Modules (ESM)



* Reusable, encapsulated, organized, convenient

Problems?

ESM for node it not support yet

How do thay work in the browser?

ESM for bowser is very very slow. When you’re using that syntax, the browser has to basically read top down. It needs to start at import statement.

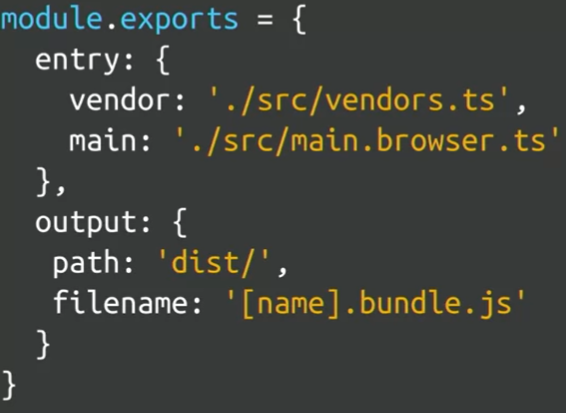
1. Introducing Webpack

Webpack is a module bundler. Let you write any module format, complies them for the browser.

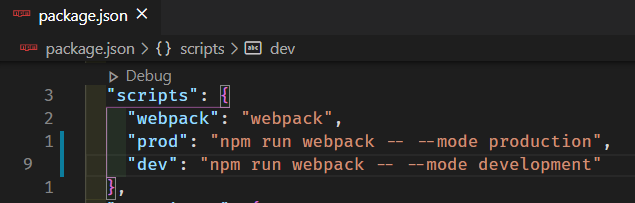
Webpack support code splitting or static async bundling, where you can create separately lazy lot of bundles at build time.

Today, it is the most performance way, if you want to write the most performant application. It allows us to be incredibly flexible or do any sort of custom workflow and makes it possible.

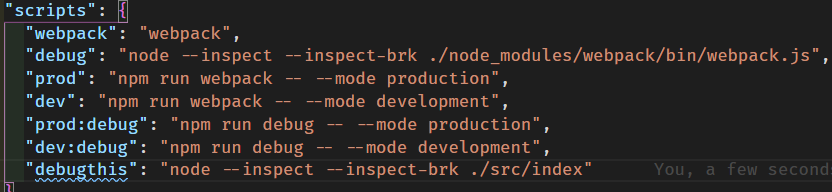
1. Configuring Webpack



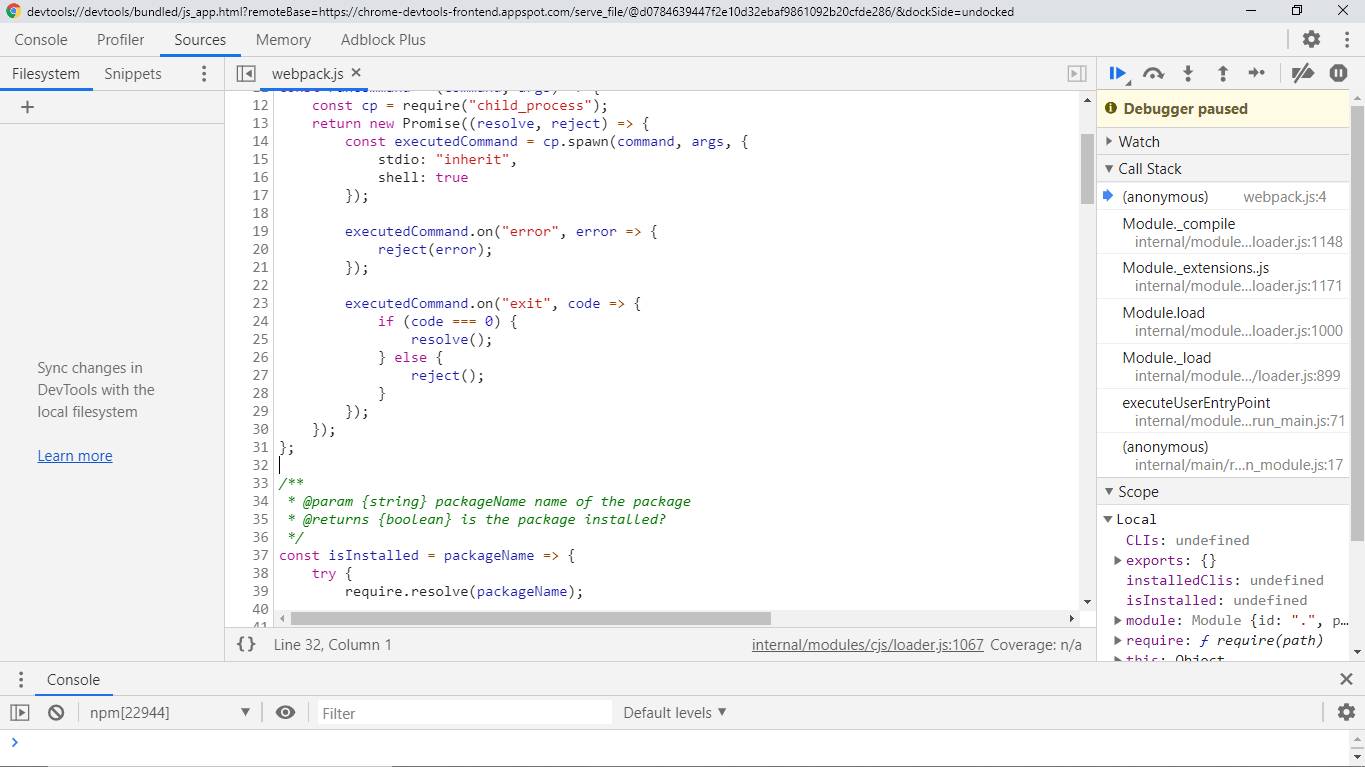
1. Webpack from scratch
2. Adding npm Scripts for Environment Builds



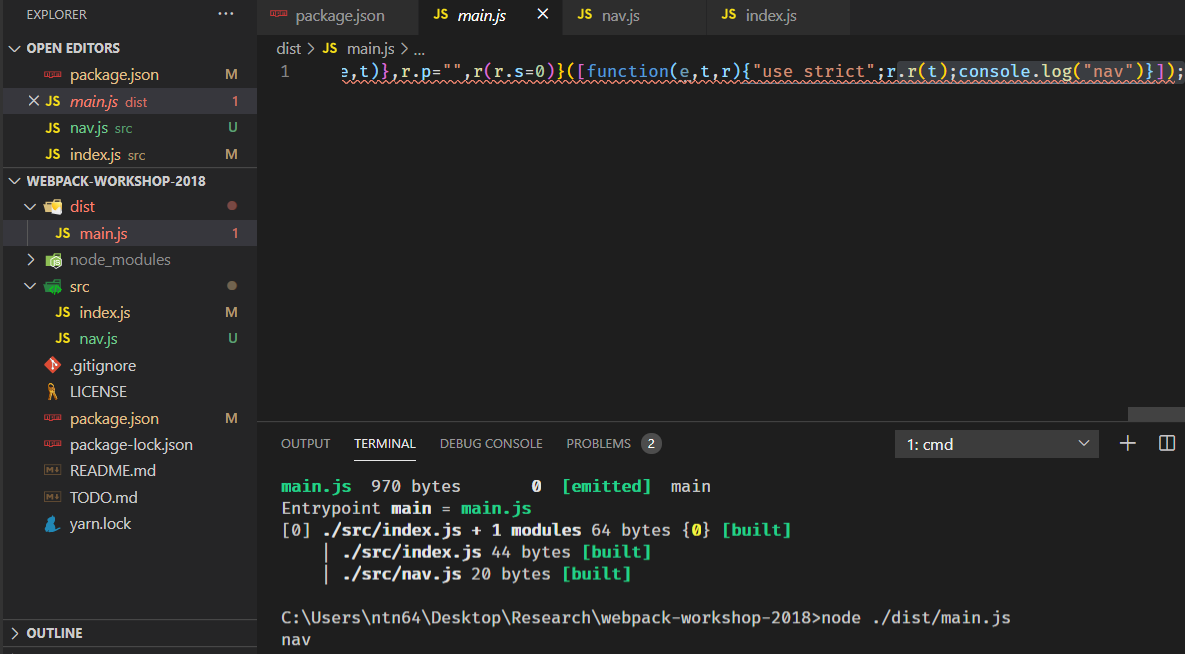
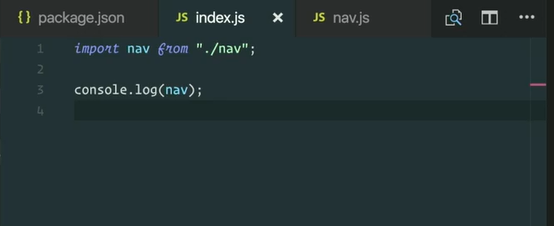
1. Setting Up Debugging



<chrome://inspect/#devices>

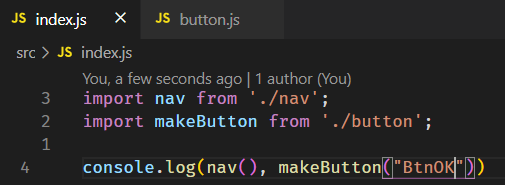
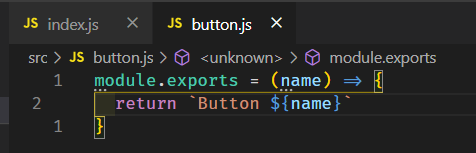


1. Coding Your First Module



1. CommonJS Export

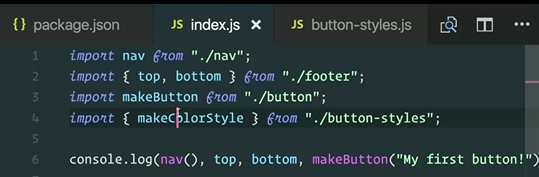
You cannot use CommonJS and ES syntax in the same file.



You should not to use CommonJS as much as possible. There are tools like Babel and Typescript that safety default to converting your ESM to CommonJS behind the scenes and then passing it to Webpack.

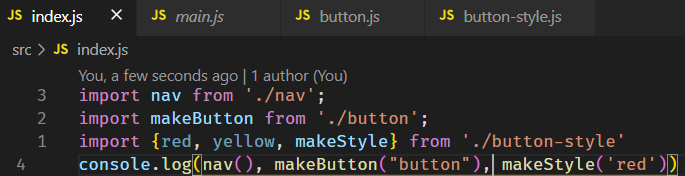
But webpack supports ESM out of the box and that’s makes it possible to tree shake, to code eliminate and all the optimizations.

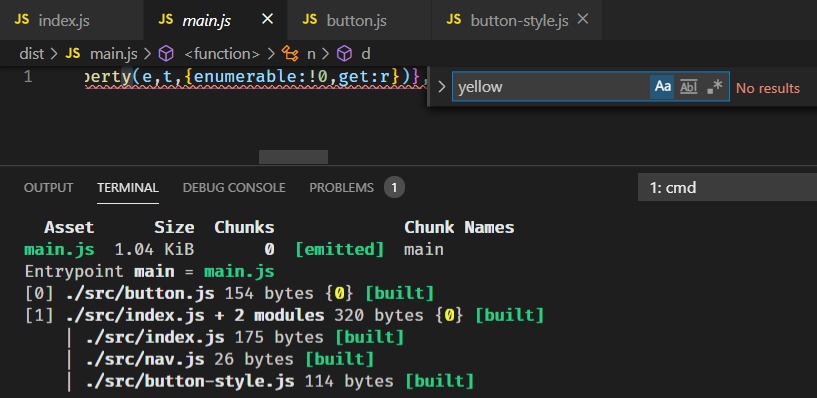
1. CommonJS Named Exports



1. Tree Skaking

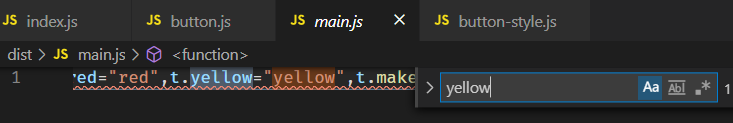
Let’s bundle this code in production.



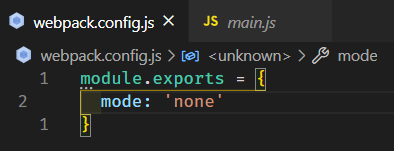
* 

This is what is called dead code elimination or tree shaking. It’s the fact that Webpack is using statically the syntax to identify what am I using?

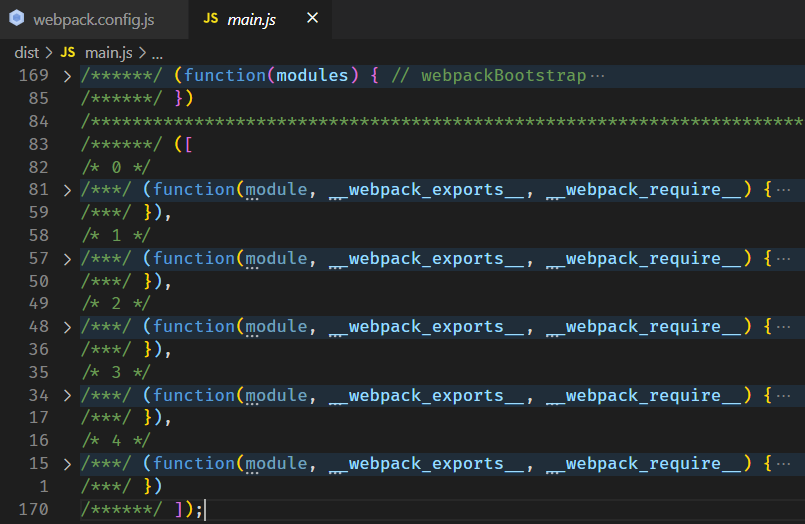
Becarefull



1. Webpack Bundle Walkthrough

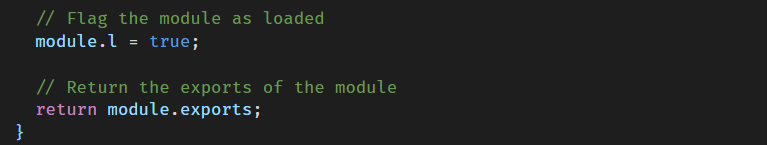


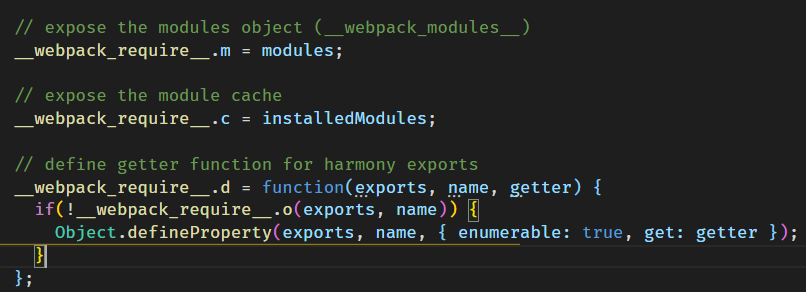
Let’s bundle code in production.



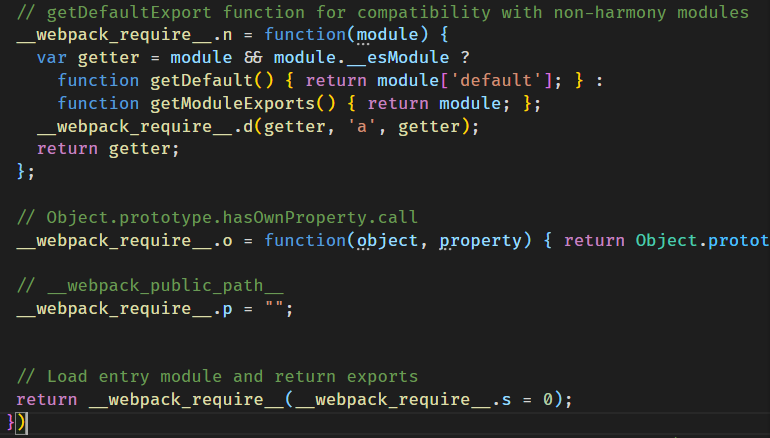
* It’s an IIFE. Take an arguments is modules -> array of IIFE

I usually call it the runtime code



It takes an ID, and it actually calls the module and then returns any exports if it’s available

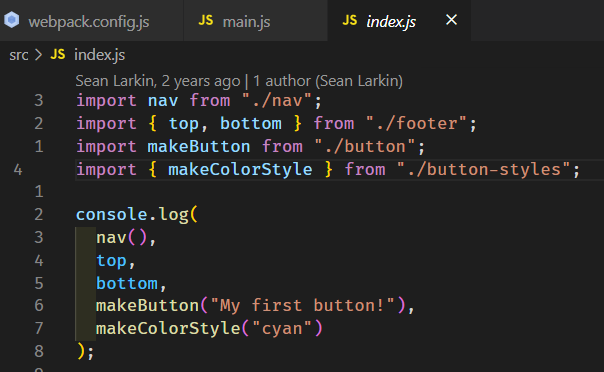
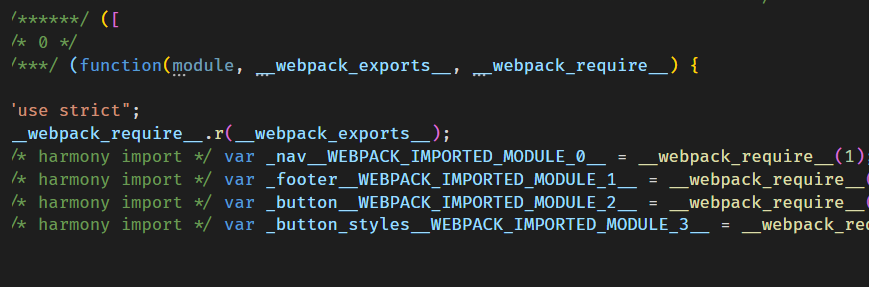
ESM support live binding, it’s a feature support cyclical dependencies. This is the code required to be able to implement it. Basically it freezing the object so that it can’t be reassigned or changed



The last line that gets executed, what happening here?

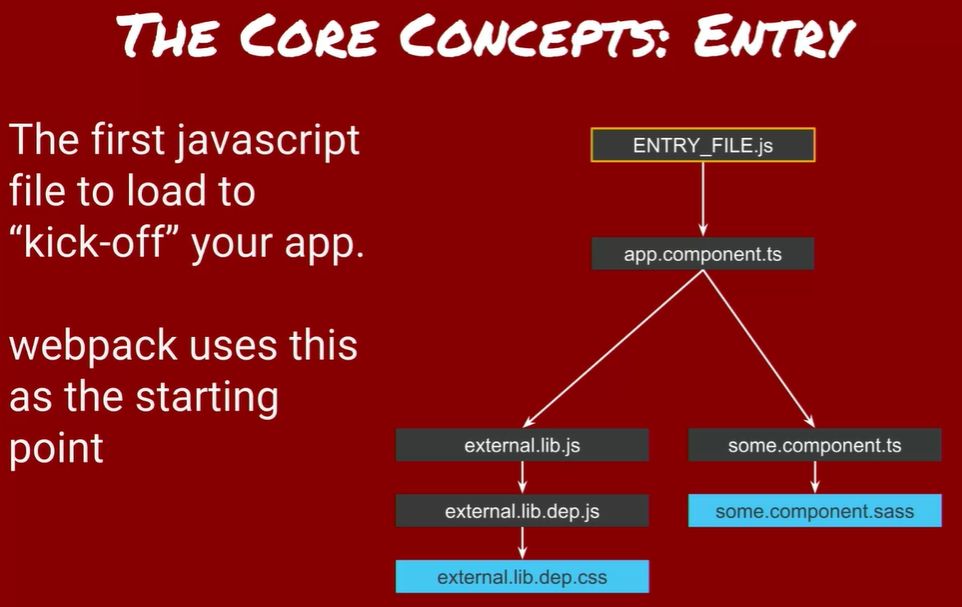
What’s the ID getting passed through? -> The first module

We’re executing our entry point.

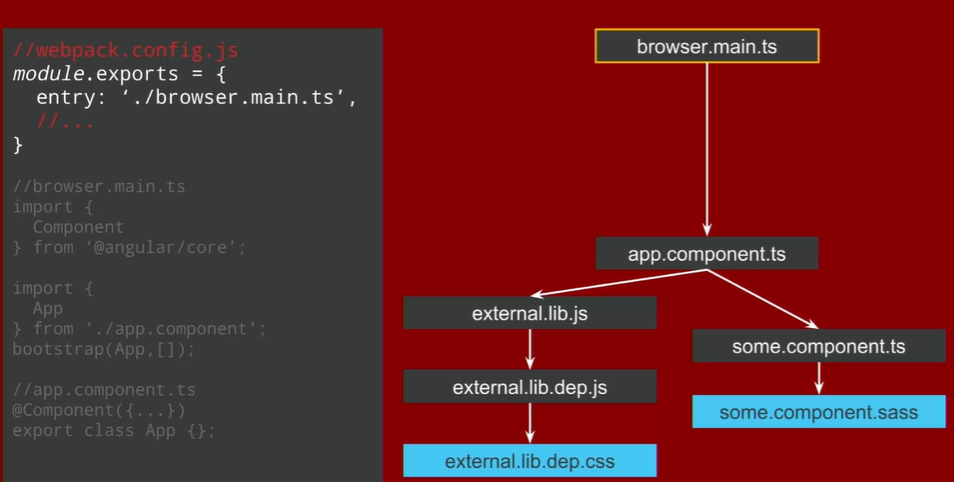


So, webpack\_require is just replacing these import statements, the require statements to st that actually work in the browser. Execute and behaves the same in the right order

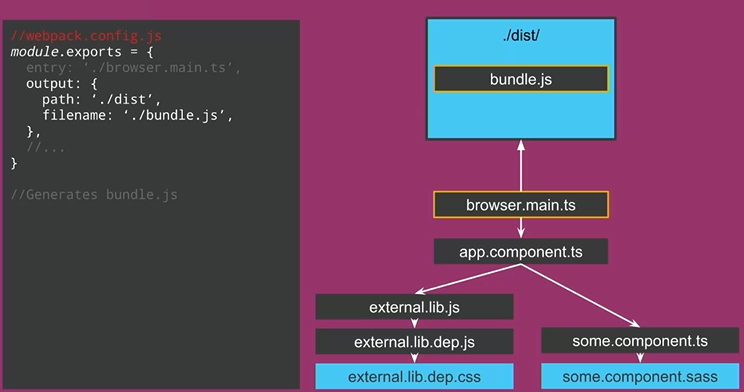
1. Webpack Core Concepts
2. Webpack Entry



The top file is your entry point. Tell webpack what (files) to load for the browser. Compliments the Output property.



1. Output & Loaders



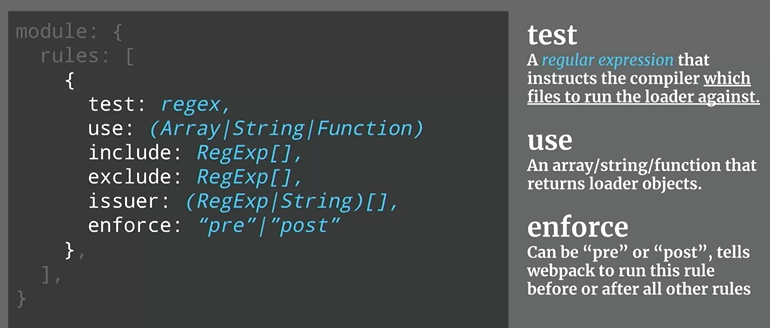
Output tell Webpack where and how to distribute bundles (compilations). Works with Entry

\***Loaders**



As webpack’s creating this dependency graph, it’s adding files to it. If webpack comes across something that matches one of these regular expressions here, this rule apply this node module, which is just a function behind the scenes and tranform whatever file that comes across.

Example: this entry file might be importing a .ts file. Webpack’s not gonna understand typescript out of the box -> it would throw out errors trying to parse that file. Ts-loader tranform .ts file to .js



1. Chaining Loaders



Style(css(less())): style.less -> style.css -> \*.js -> inlineStyleBrowser.js

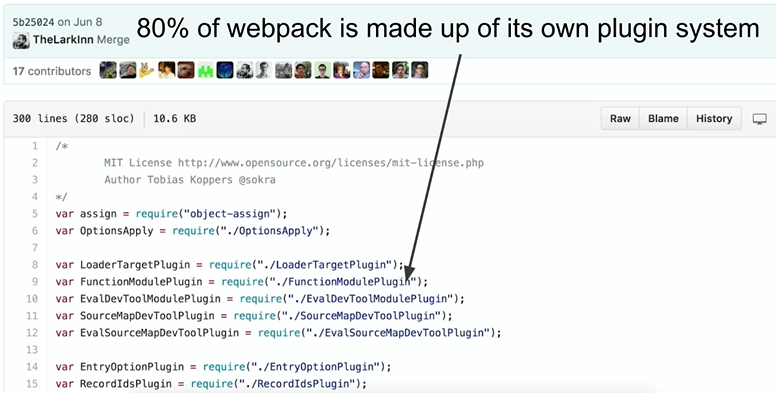
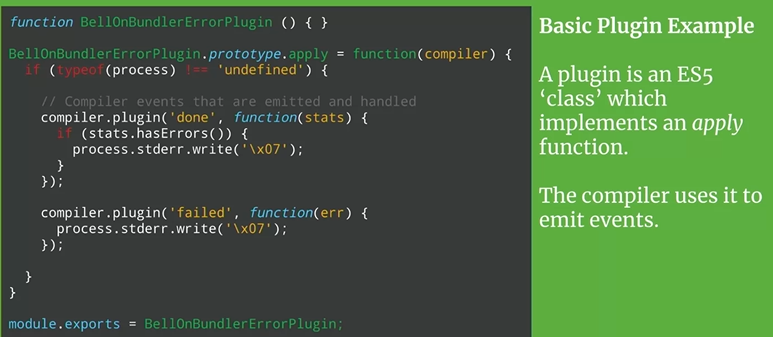
1. Webpack Plugins

Plugins: adds additional funcitonality to Compilations(optimized bundled modules). More powerful more access to CompilerAPI. Does everything else you’d ever want to in webpack.

+ Objects (with an ‘apply’ property): it’s a js object has an apply property in the prototype chain

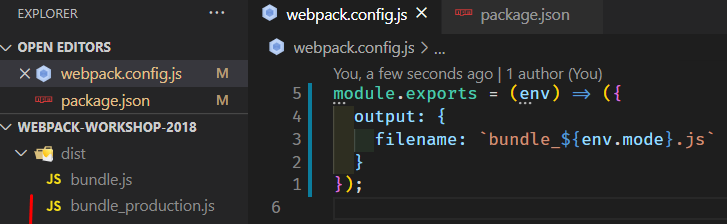
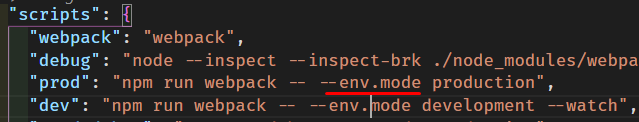
+ Allow you to hook into the entire compilation lifecycle

+ Webpack has a variety of built in plugins

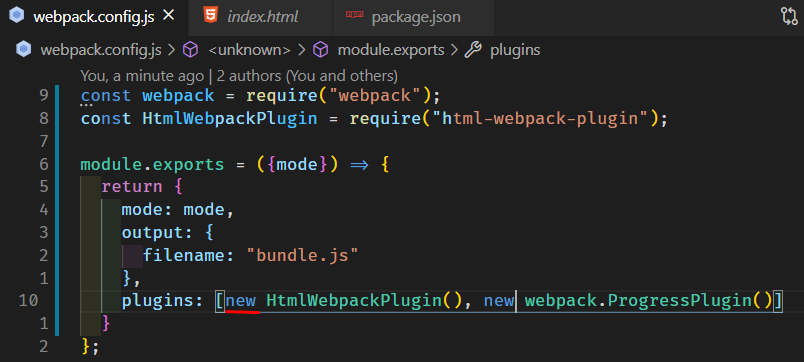


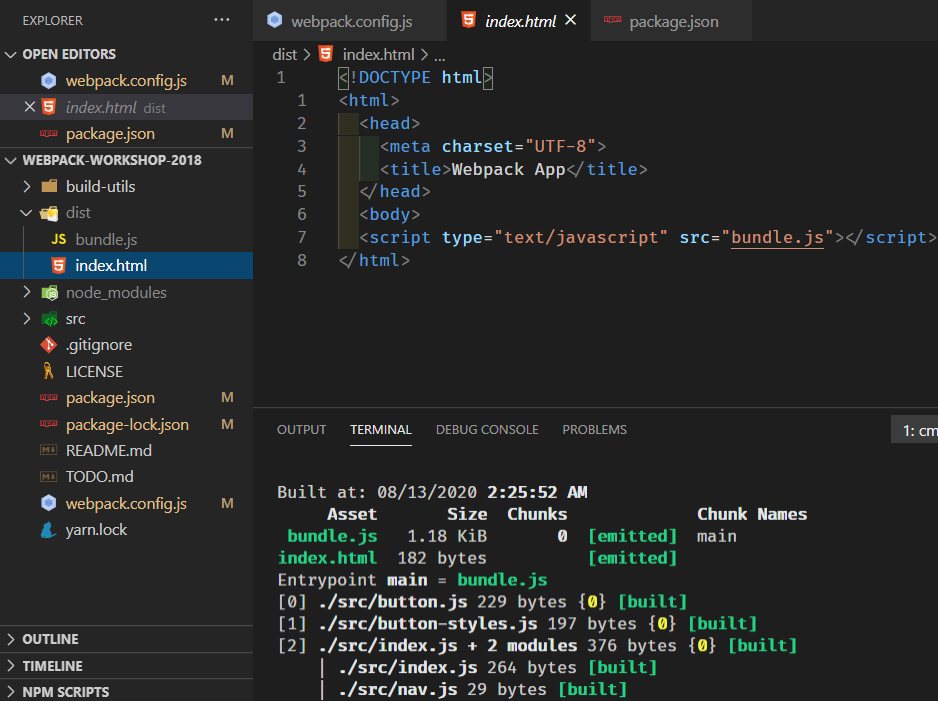
Webpack itself is a completely event driven architecture. It allow us to pivot really quickly. We could instantly adopt a new feature without breaking anything. Or we can drop a feature really easily and we could slowly deal with just a deprecation and barely have to manage it without breaking changes.

1. Passing Variable to Webpack Config



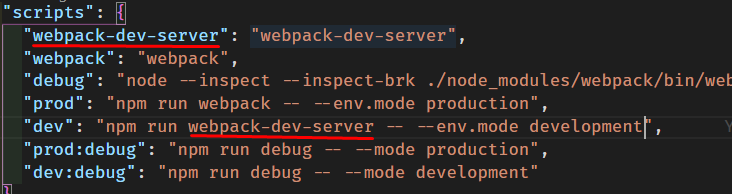
1. Adding Webpack Plugins





Html-webpack-plugin is an essential, specifically for single page applications. It inject whatever output assets are there into this file for you

1. Setting Up a Local Development Server



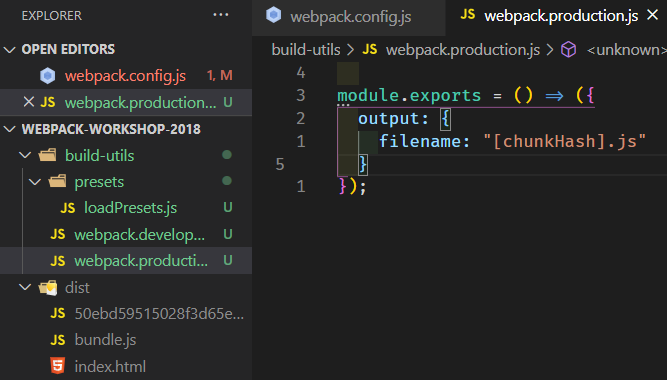
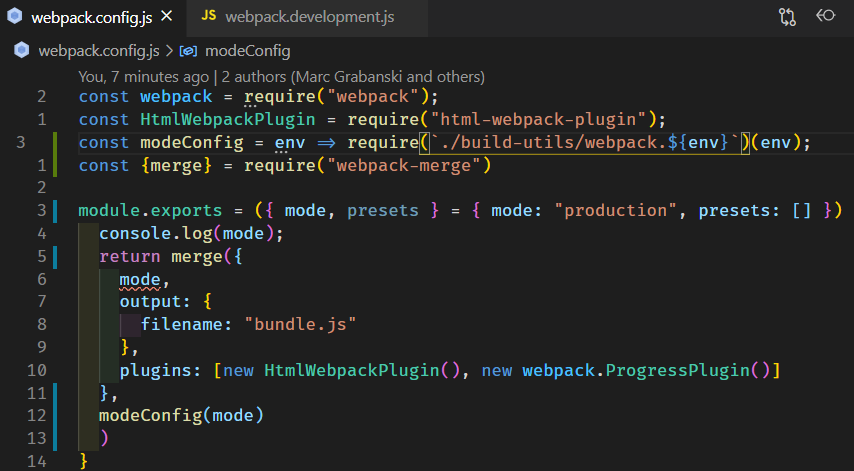
I wanna talk about webpack-dev-server. It takes the whole contents of your disk and server it up.

It’s going to reset and refresh the browser any time that we make changes. So this is one of these awesome opportunities using the dependency graph

It is a web server based on Express. And all it’s doing is webpack, instead of creating a bundle to your dist folder. It actually generates a bundle in memory and it serves that information up to Express which then does a web socket connection and says, hey, I just updated. Then it reload.

1. Splitting Environment Config Files

Libary webpack-merge, it’s essentially just object assign for webpack-config



1. Webpack Q&A

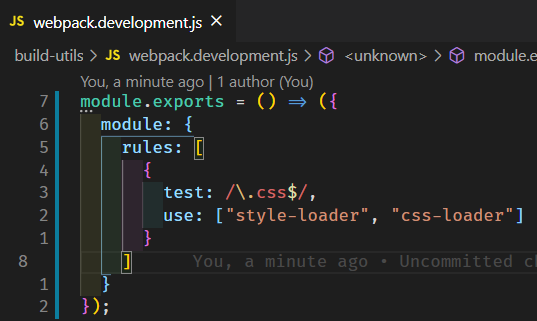
So you can use webpack-dev-server for server side for development?

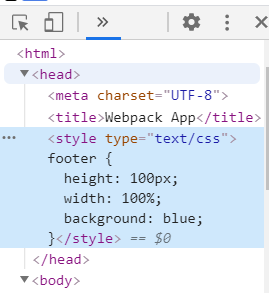
* You can use it, but you actually use a pieace that abstracts it called webpack-dev-middleware. So webpack-dev-server is just made up of webpack-dev-middleware and express. So you can use this package itself standalone. You can npm install webpack-dev-middleware and add it to your express application.

Are there situations where webpack, the process runs into an out of memory error and where would you capture that exception?

* When you think about in just programmatic complexity, webpack has to create an entire graph of all of your source files. And store it in memory before it creates a bundle. webpack space complexity, will be linear in terms of how many modules you have in your app. So you will end up consuming more and more memory because it needs more memory. So if it's for a production build and it's just a build and not dev server, I would say increasing the memory limit for Node. But we have some caching for webpack 5. We have a whole caching store that's aiming to solve this. So that would be one way to cache it.

1. Using Plugins
2. Using CSS with Webpack

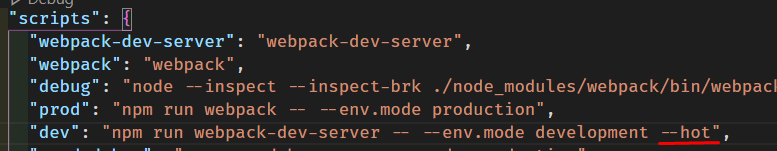




This is what style-loader actually does. It’s adds the script tag in the browser.

1. Hot Module Replacement with CSS

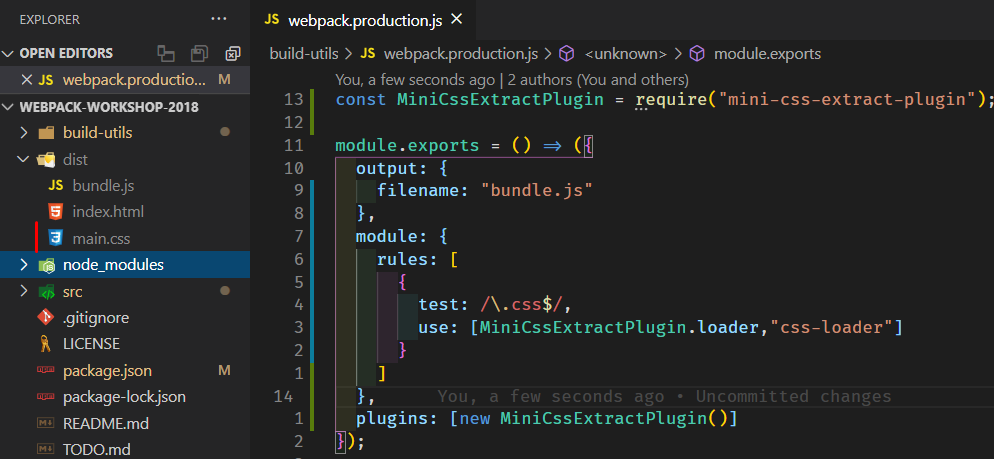
Loaders are really useful for helping support a unique webpack feature called hot module replacement.

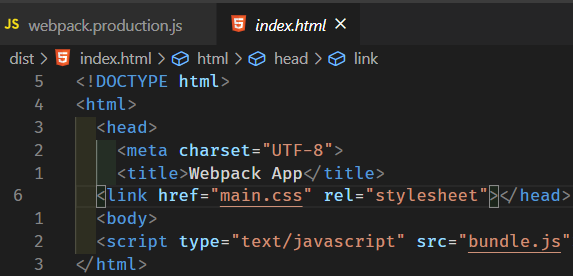
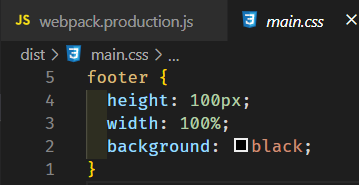


If we change something. Instantly, you’re seeing changes and we’re not reloading the browser.

So the CSS that we have now, it’s just adding a module and it’s blocking the main thread. Because you’re relying on JS to attach a style tag.

* We would wanna extract it out and have it in a seperate tag. We can do it with **mini-css-extract-plugin**





1. File Loader & URL Loader

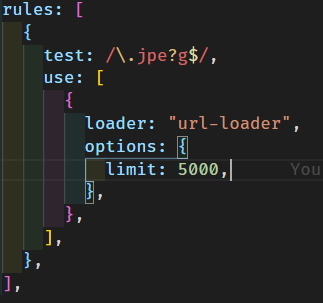


Why do we only pass node module references for these loaders?

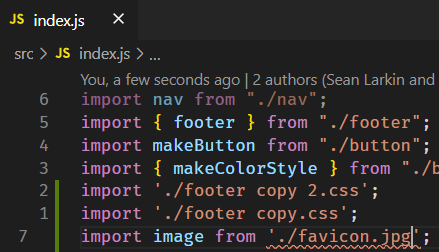
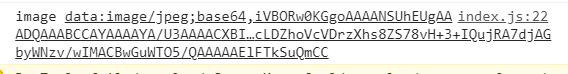
* Loaders are serialized, the first premise with webpack is that when loaders were originally invented, the way that you were using them actually was doing something like this.

With css-loader: import {style} from ‘css-loader!./button.css’

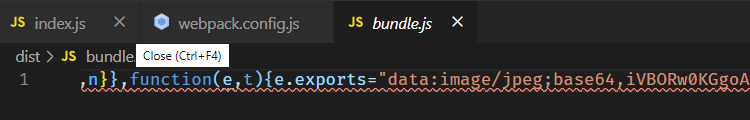
What happens is that we stringtify and we serialize this entire request, but sometimes you can add options and even functions as options, those aren’t serializable. We couldn’t just pass a function itself in the config or the function object cuz we need to be able to serialize that information and then we parallelize it.



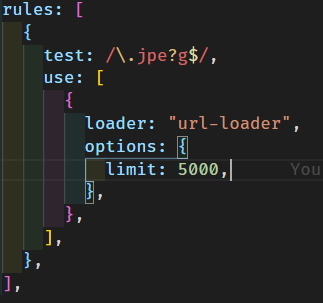
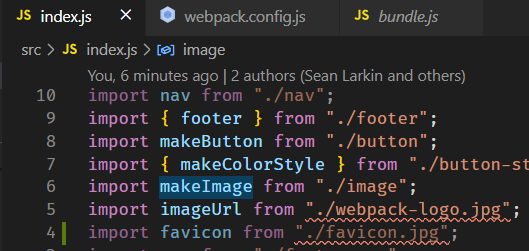
1. Loading Images with JavaScript

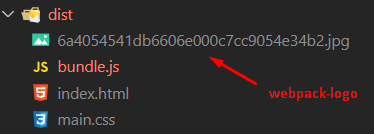
 

It will increase bundle file size, not good for performance. In production:



1. Limit Filesize Option in URL Loader

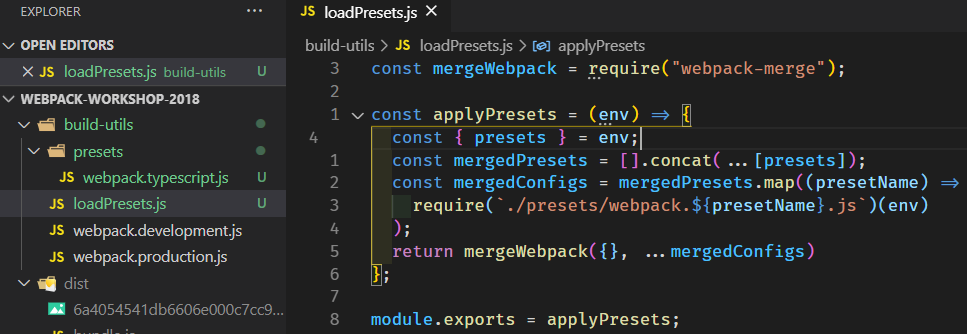
 

 (webpack-logo.jpg – 162KB)

This option specifying the maximum size of a file in bytes. If the file size is **equal** or **greater** than the limit [**file-loader**](https://webpack.js.org/loaders/file-loader/) will be used (by default) and all query parameters are passed to it.

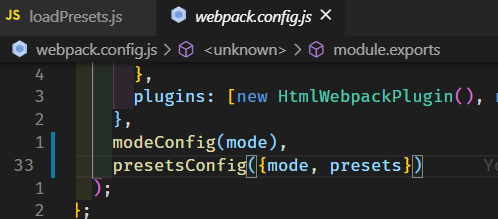
1. Implementing Presets

The idea of presets is that, there are gonna be more than just dev and prod. You’ve gonna have some different scenarios like, I wanna try this feature or I wanna analyze my build just this one time. But you don’t want it like shipped right in your prod config because it’s not relevant every time you run it. (can call it add-ons)



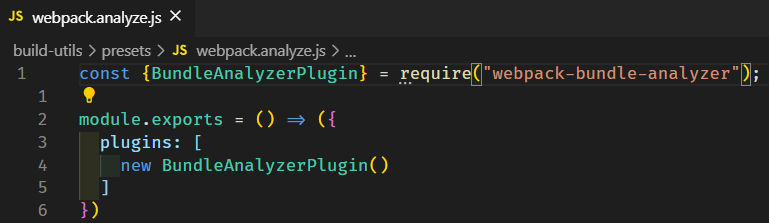
In package.json add:

"prod:typescript": "npm run prod -- --env.presets typescript",

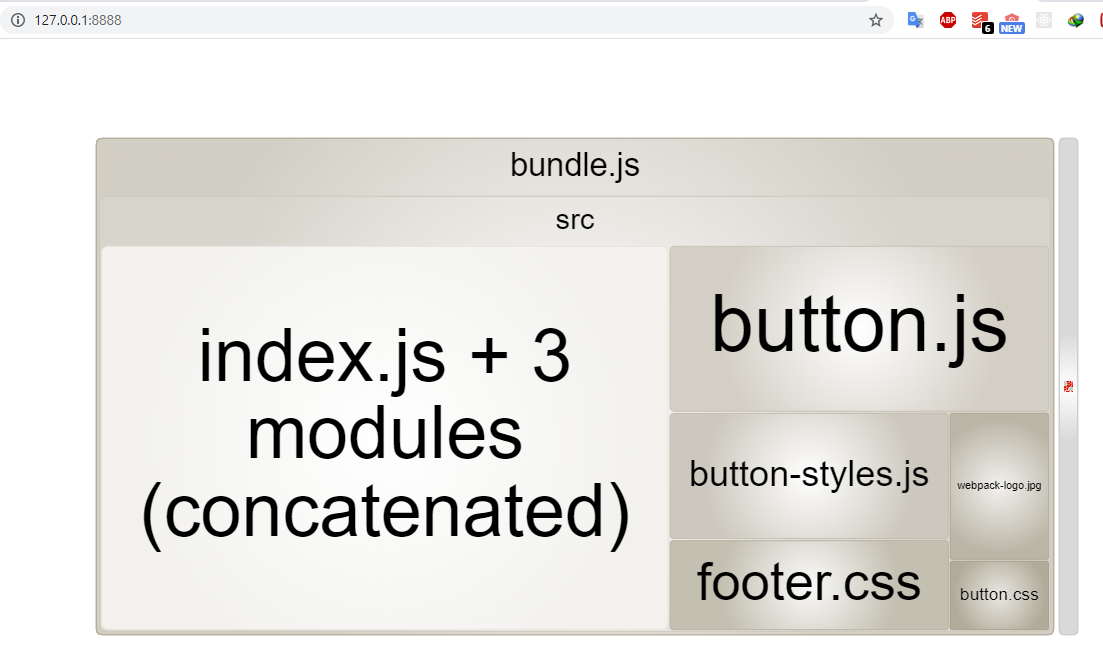


1. Bundle Analyzer Preset

Webpack-bundle-analyzer analyzing why did this certain dependency get pulled into my application? Why is this file so large?



    "prod:analyze": "npm run prod -- --env.presets analyze",

* 

This is incredibly valued for identifying like, why do I have duplication across these bundles? Or why is this file not getting separated out? Or why is this not being tree shaken.

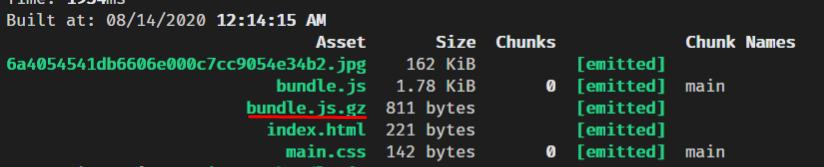
We’ll use this preset for identifying performance issues.

1. Compression Plugin

Compression-webpack-plugin take any of your assets that you wanna include, and it compresses them.

"compress": "npm run prod -- --env.presets compress",

* Npm run compress -- --env.presets analyze



If you wanted to take performance seriously. I’ve seen this method and I wanna create a service around this. Allows you to have an AV testing enviroment for different types of methododology, different ways of bundling your application and switching on these different presets in combination.

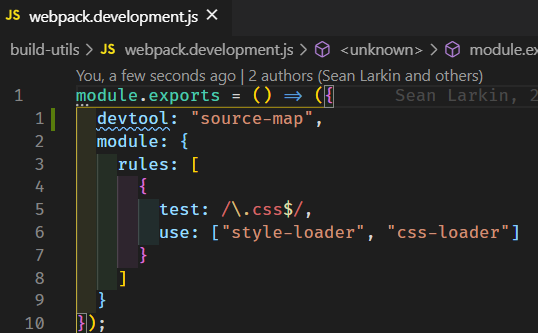
Webpack-bundle-analyzer:

if you set the analyzer mode property to static, what this does is it generates all the static assets to visualize. And then they actually serve it up on a separate port for every production build that they do. So, at any time they notice we've regressed in performance.

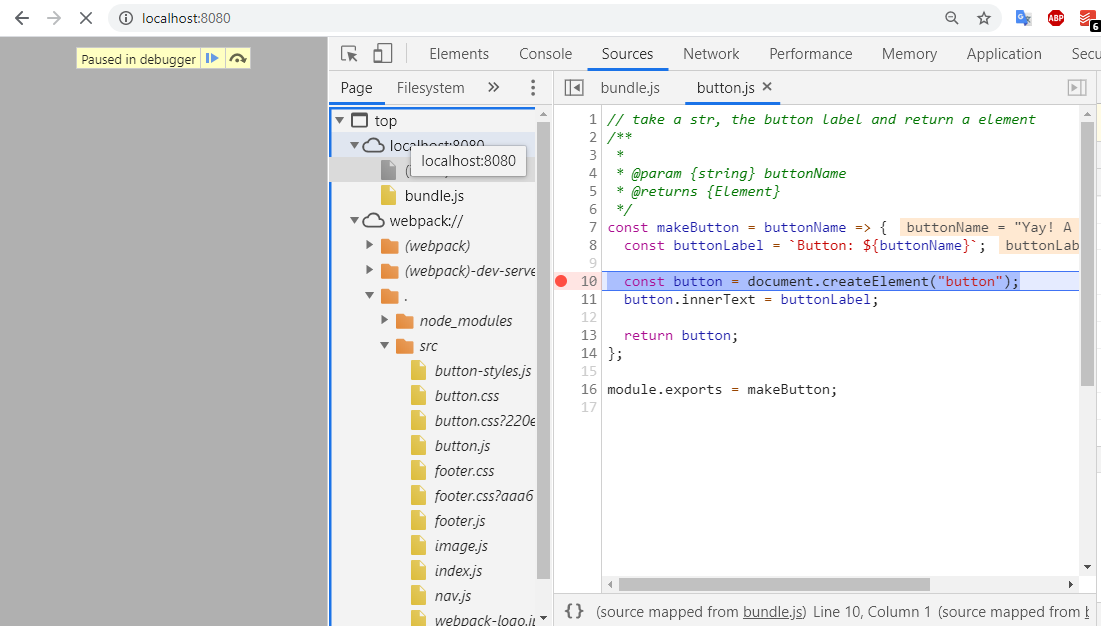
For any build they can just go to like colon, whatever port forward slash analyze, and it's there for that build. And so, they just redeploy it every time as a part of their build.

1. Source Maps

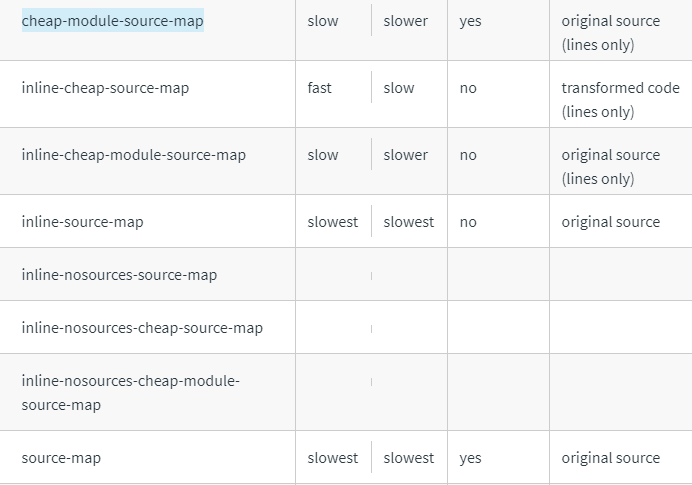
<https://webpack.js.org/configuration/devtool/>



You can see your entire project structure if you want.



You can use another devtool for better speeded builds



1. Wrapping Up

Is there a lazy load plugin you'd recommend?

* So lazy loading is code splitting in Webpack.

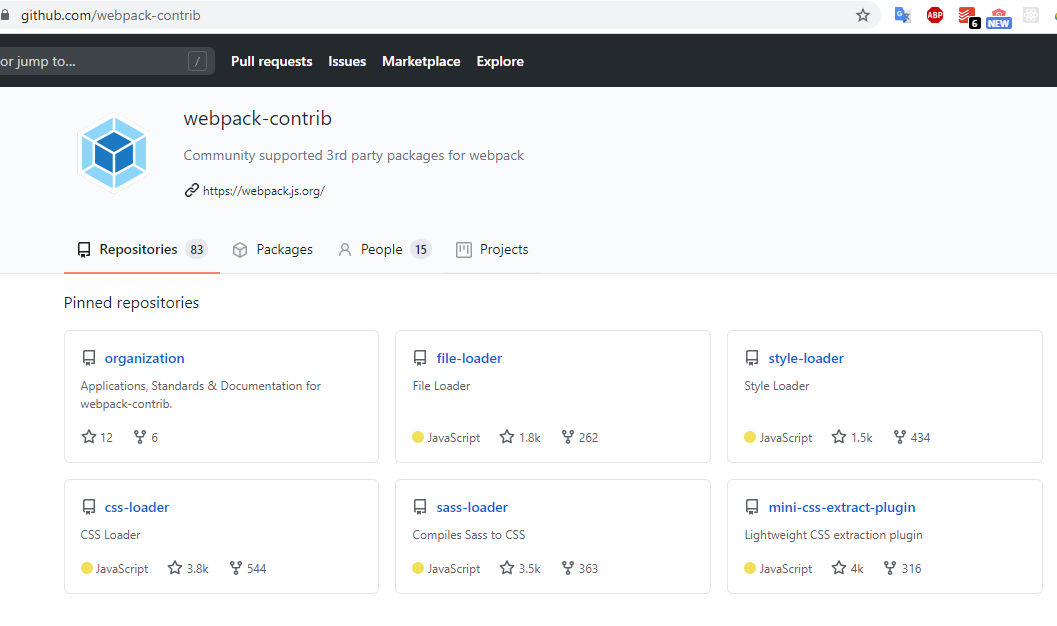
For example: I would only want to load the footer when somebody clicks on that button.



This syntax only work with typescript or babel.

You don’t need any plugin for it.

How do you go about finding good plugins versus bad plugins?



I would say the best way is just to look and see, kind of like, how maintained is it? What version of Webpack is in their tests?

So really the best way is kinda just like the same way you would approach anything else, how much is it contributed? How active are its commits? And it may just be, it's a finished module.