Webpack

# Iefe

First in order to maintain scope for each file we used IEFEs but it had its problems:

1. Slowness
2. Dead code and unused code
3. No way to break or lazy load

Common.js

CommonJS is a module formatting system. It is a standard for structuring and organizing JavaScript code. CJS assists in the server-side development of apps and it’s format has heavily influenced NodeJS’s module management.

Without module systems like CommonJS, dependancies had to be loaded in <script> tags in the header of an HTML file, OR all code had to be lumped together which is incredibly slow and inefficient for file loading.

CommonJS wraps each module in a function called ‘require’, and includes an object called ‘module.exports’, which exports code for availability to be required by other modules. All you have to do is add whatever you want accessible to other files onto the ‘exports’ object and require the module in the dependent file. the syntax for the require function is **“var VariableName = require(‘moduleId\_or\_pathToModule’);”**.

Text

Description automatically generated

Modules are loaded synchronously, so modules that are dependent on other modules must be read further down in the code.\

## Cons

* There is no browser support for commonjs
* Problems with circular dependencies
* Either use a server to translate CJS modules to something usable in the browser.
* Or use XMLHttpRequest (XHR) to load the text of modules and do text transforms/parsing in browser.

# ESM (Eschma script module

A module is a function or group of similar functions. They are grouped together within a file and contain the code to execute a specific task when called into a larger application.

1. **Independent/Self-contained:** A module has to be as detached from other dependencies as possible.
2. **Specific:** A module needs to be able to perform a single or a related group of tasks. The core essence of creating them in the first place is to create separate functionalities. One module, one (kind of) task.
3. **Reusable:** A module has to be easy to integrate into various kinds of programs to perform its task.

A module in JavaScript is just a file containing related code.

In JavaScript, we use the import and export keywords to share and receive functionalities respectively across different modules.

* The export keyword is used to make a variable, function, class or object  accessible to other modules. In other words, it becomes a public code.
* The import keyword is used to bring in public code from another module.

Text

Description automatically generated

## Imports

### Named imports and exports

Import { abc } from ‘./file’

### Default imports and exports

Import anything from ‘./file’

### Dynamic imports

Why?

* We cant dynamically set import name

import ... from getModuleName(); // Error, only from "string" is allowed

* We cant conditionally import a module into another

Graphical user interface, text, application

Description automatically generated

The import(module) expression loads the module and returns a promise that resolves into a module object that contains all its exports. It can be called from any place in the code.

Graphical user interface, text

Description automatically generated

## Cons

Modules are very slow in the browser because the browser at runtime needs to resolve the modules and get the exports and so on

# Webpack

Is a module bundler

Write any module format, require or use any format and allows you to use in the browser

Supports static async bundling to use lazy loading

## How bundling process works

1. Each file is a module (js, ts, css, html …)
2. When we import files into each other we create a dependency graph
3. During the bundling process, modules are combined into chunks.
4. Chunks combine into chunk groups and form a graph (ChunkGraph) interconnected through modules. When you describe an entry point - under the hood, you create a chunk group with one chunk.

Chunk Graph

Chunk groups

chunks

modules

* When we have one entry point -> it creates one chunk group containing one chunk
* When we have more than one entry point for example

Text

Description automatically generated

Two chunk groups with names home and about are created. Each of them has a chunk with a module - ./home.js for home and ./about.js for about

* When we use dynamic imports a non-initial chunk is created for that module

Text

Description automatically generated

Initial chunk with name main is created. It contains:

./src/index.jsx

react

react-dom

and all their dependencies, except ./app.jsx

Non-initial chunk for ./app.jsx is created as this module is imported dynamically.

By default, there is no name for non-initial chunks so that a unique ID is used instead of a name. When using dynamic import we may specify a chunk name explicitly by using a ["magic" comment](https://webpack.js.org/api/module-methods/#magic-comments):

Text

Description automatically generated

## Chunks

* initial is the main chunk for the entry point. This chunk contains all the modules and their dependencies that you specify for an entry point.
* non-initial is a chunk that may be lazy-loaded. It may appear when [dynamic import](https://webpack.js.org/guides/code-splitting/#dynamic-imports) or [SplitChunksPlugin](https://webpack.js.org/plugins/split-chunks-plugin/) is being used.

Each chunk has a corresponding **asset**. The assets are the output files - the result of bundling.

## Config

Webpack looks for entry property and defaults to src/index.js if nothing is configured

## Modes

Providing the mode configuration option tells webpack to use its built-in optimizations accordingly.

string = 'production': 'none' | 'development' | 'production'

Graphical user interface, application

Description automatically generated

If we are running on –watch and we change any code that isn’t related to our entry point then webpack doesn’t make any changes to the bundle

Webpack looks for config file of name webpack.config.js

Graphical user interface, text, application

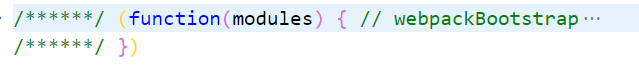
Description automatically generated

The output will be

Text

Description automatically generated

The generated main.js file will be an IEFE function that takes modules as parameter



The modules are passed as object of IEFEs

The key will be the file path and value is an IEFE of the file contents

Text

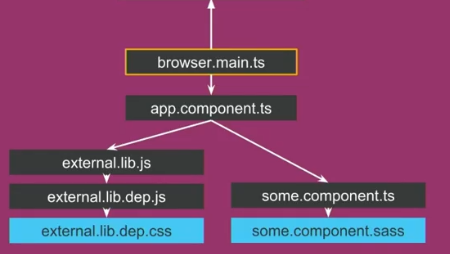
Description automatically generated

## Dead code elimination (tree shaking)

If we bundle the code in production and search for an export that wasn’t imported anywhere, we wont find it, because in production it does tree shaking which is useful when you only use one function from a large library then we don’t need to bundle the entire size of the library

## Entry point

It’s the root of the files which is processed by webpack



Entry point by default is ./src/main.ts

entry: './path/to/my/entry/file.js',

is shorthand for   
entry: {

main: './path/to/my/entry/file.js',

},

### Multi main entry

This is useful when you would like to inject multiple dependent files together and graph their dependencies into one "chunk".

entry: ['./src/file\_1.js', './src/file\_2.js'],

both files will be added into one bundle

this is useful when we have dependencies from vendors as bootstrap or jquery

### different entry points

Text

Description automatically generated

This will create a separate bundle for each file

We are telling webpack that we would like 3 separate dependency graphs

### specifying entry points options

* dependOn: The entry points that the current entry point depends on. They must be loaded before this entry point is loaded.
* filename: Specifies the name of each output file on disk.
* import: Module(s) that are loaded upon startup.
* library: Specify [library options](https://webpack.js.org/configuration/output/#outputlibrary) to bundle a library from current entry.
* runtime: The name of the runtime chunk. When set, a new runtime chunk will be created. It can be set to false to avoid a new runtime chunk since webpack 5.43.0.
* publicPath: Specify a public URL address for the output files of this entry when they are referenced in a browser. Also, see [output.publicPath](https://webpack.js.org/configuration/output/#outputpublicpath).
* chunkLoading: false: Disable chunks that are loaded on demand and put everything in the main chunk.

Text

Description automatically generated

Text

Description automatically generated

### Dynamic entry

If a function is passed then it will be invoked on every [make](https://webpack.js.org/api/compiler-hooks/#make) event. make event triggers when webpack starts and for every invalidation when [watching for file changes](https://webpack.js.org/configuration/watch/).

Graphical user interface, text

Description automatically generated

### caveats

* runtime and dependOn should not be used together on a single entry, so the following config is invalid and would throw an error
* Make sure runtime must not point to an existing entry point name
* dependOn must not be circular

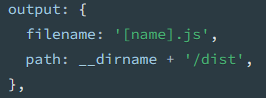
A rule to consider: one entry point per HTML page. SPA: one entry point, MPA: multiple entry points.

## Output

Where is the output file destination path and what is its name

while there can be multiple entry points, only one output configuration is specified.

We can use substitutions in filename



### asyncChunk

Create async chunks that are loaded on demand.

## Loaders and Rules

Loaders transform files that are added to the dependency graph

Webpack by itself only knows javascript, so when we want it to pack any other type of resources like **.css** or **.scss** or **.ts**, webpack needs help in order to compile and bundle those non-javascript types of resources.  
  
Loaders are the node-based utilities built for webpack to help webpack to compile and/or transform a given type of resource that can be bundled as a javascript module.

Loaders are transformations that are applied to the source code of a module. They allow you to pre-process files as you import or “load” them.

For example:

**css-loader** is the npm module that would help webpack to collect CSS from all the css files referenced in your application and put it into a string.

And then **style-loader** would take the output string generated by the above css-loader and put it inside the <style> tags in the index.html file.

Another example:

Using babel-loader to transform .js files from es6 syntax to es5 syntax

### Rule set

It used a loader based on regex matching the file name

Text

Description automatically generated

Example

Text

Description automatically generatedText

Description automatically generated

### Note

Loaders always execute from right to left

## Plugins

To add extra functionality to webpack

It’s a es5 class or object that has apply property in prototype chain

Allows you to hook into webpack lifecycle

Text

Description automatically generated

Here we plugin to the done and failed events of webpack

Text

Description automatically generated

## Modules

Graphical user interface, text, application

Description automatically generated

## Manifest

In a typical application or site built with webpack, there are three main types of code:

1. The source code you, and maybe your team, have written.
2. Any third-party library or "vendor" code your source is dependent on.
3. A webpack runtime and **manifest** that conducts the interaction of all modules.

### Runtime

The runtime, along with the manifest data, is all the code webpack needs to connect your modularized application while it's running in the browser. It contains the loading and resolving logic needed to connect your modules as they interact. This includes connecting modules that have already been loaded into the browser as well as logic to lazy-load the ones that haven't

### Manifest

Once your application hits the browser in the form of index.html file, some bundles and a variety of other assets required by your application must be loaded and linked somehow. That /src directory you meticulously laid out is now bundled, minified and maybe even split into smaller chunks for lazy-loading by webpack's [optimization](https://webpack.js.org/configuration/optimization/). So how does webpack manage the interaction between all of your required modules? This is where the manifest data comes in.

As the compiler enters, resolves, and maps out your application, it keeps detailed notes on all your modules. This collection of data is called the "Manifest," and it's what the runtime will use to resolve and load modules once they've been bundled and shipped to the browser. No matter which [module syntax](https://webpack.js.org/api/module-methods) you have chosen, those import or require statements have now become \_\_webpack\_require\_\_ methods that point to module identifiers.

Using the data in the manifest, the runtime will be able to find out where to retrieve the modules behind the identifiers.

The runtime will do its thing, utilizing the manifest, and everything will appear to magically work once your application hits the browser. However, if you decide to improve the performance of your projects by utilizing browser caching, this process will all of a sudden become an important thing to understand.

By using content hashes within your bundle file names, you can indicate to the browser when the content of a file has changed, thus invalidating the cache. Once you start doing this though, you'll immediately notice some funny behavior. Certain hashes change even when their content apparently does not. This is caused by the injection of the runtime and manifest, which changes every build.

## Hot Module Replacement

Hot Module Replacement (HMR) exchanges, adds, or removes [modules](https://webpack.js.org/concepts/modules/) while an application is running, without a full reload. This can significantly speed up development in a few ways:

* Retain application state which is lost during a full reload.
* Save valuable development time by only updating what's changed.
* Instantly update the browser when modifications are made to CSS/JS in the source code, which is almost comparable to changing styles directly in the browser's dev tools.

HMR can be used in development as a LiveReload replacement. [webpack-dev-server](https://webpack.js.org/configuration/dev-server/) supports a hot mode in which it tries to update with HMR before trying to reload the whole page.

## Target

Because JavaScript can be written for both server and browser, webpack offers multiple deployment targets that you can set in your webpack [configuration](https://webpack.js.org/configuration).

## Optimisations

### Chunk id

Tells webpack which algorithm to use when generating chunk ids.

Supported values are:

* Natural: numeric ids in order of usage ( mode is none)
* Named: readable ids for debugging (default in development environment)
* Deterministic: Short numeric ids which will not be changing between compilation (default in production environment)
* Size: Numeric ids focused on minimal initial download size
* Total-size: numeric ids focused on minimal total download size
* False: tells webpack that none of built-in algorithms should be used, as custom one can be provided via plugin

Text

Description automatically generated

## concatenateModules

Tells webpack to find segments of the module graph which can be safely concatenated into a single module

By default optimization.concatenateModules is enabled in production [mode](https://webpack.js.org/configuration/mode/) and disabled elsewise.

## flagIncludedChunks

Tells webpack to determine and flag chunks which are subsets of other chunks in a way that subsets don’t have to be loaded when the bigger chunk has been already loaded. By default optimization.flagIncludedChunks is enabled in production mode and disabled elsewise.

## mangleExports

controls changing the names of exports

allowed values:

* Size: Short names - usually a single char - focused on minimal download size.
* Deterministic: Short names - usually two chars - which will not change when adding or removing exports. (default in production mode)
* True: Same as 'deterministic'
* False: Keep original name. Good for readability and debugging

### mergeDuplicateChunks

Tells webpack to merge chunks which contain the same modules

## minimize

Tell webpack to minimize the bundle using the TerserPlugin or the plugin(s) specified in optimization.minimizer

Webpack uses Terser package for JS files minification

Text

Description automatically generated

## moduleIds

tells webpack which algorithm to use when choosing module ids. ??

allowed values:

* Natural: Numeric ids in order of usage.
* Named: Readable ids for better debugging.
* Deterministic: Module names are hashed into small numeric values.
* Size: Numeric ids focused on minimal initial download size.

## nodeEnv

Tells webpack to set process.env.NODE\_ENV to a given string value.

* any string: the value to set process.env.NODE\_ENV to.
* false: do not modify/set the value of process.env.NODE\_ENV.

### portableRecords

tells webpack to generate records with relative paths to be able to move the context folder.

By default optimization.portableRecords is disabled. Automatically enabled if at least one of the records options provided to webpack config: recordsPath, recordsInputPath, recordsOutputPath

### removeAvailableModules

ells webpack to detect and remove modules from chunks when these modules are already included in all parents. Setting optimization.removeAvailableModules to true will enable this optimization. Enabled by default in production mode.

optimization.removeAvailableModules reduces the performance of webpack, and will be disabled in production mode by default in next major release. Disable it in production mode if you want extra build performance.

### removeEmptyChunks

Tells webpack to detect and remove chunks which are empty. Setting optimization.removeEmptyChunks to false will disable this optimization.

### runtimeChunk

### sideEffects

Tells webpack to recognise the [sideEffects](https://github.com/webpack/webpack/blob/master/examples/side-effects/README.md) flag in package.json or rules to skip over modules which are flagged to contain no side effects when exports are not used.

optimization.sideEffects depends on [optimization.providedExports](https://webpack.js.org/configuration/optimization/#optimizationprovidedexports) to be enabled. This dependency has a build time cost, but eliminating modules has positive impact on performance because of less code generation. Effect of this optimization depends on your codebase, try it for possible performance wins

allowed values

A picture containing text

Description automatically generated

Example package.json

Text

Description automatically generated

### usedExports

tells webpack to determine used exports for each module.

Information collected by optimization.usedExports is used by other optimizations or code generation i.e. exports are not generated for unused exports, export names are mangled to single char identifiers when all usages are compatible. Dead code elimination in minimizers will benefit from this and can remove unused exports.

## Code splitting

This feature allows you to split your code into various bundles which can then be loaded on demand or in parallel. It can be used to achieve smaller bundles and control resource load prioritization which, if used correctly, can have a major impact on load time.

Code splitting can be done using different methods

1. multiple entry points:

if we add multiple entry points each will produce its own bundle

problems with this approach:

* If there are any duplicated modules between entry chunks they will be included in both bundles.
* It isn't as flexible and can't be used to dynamically split code with the core application logic

To fix this issue we can use

## Configuration

### Context

The base directory, an **absolute path**, for resolving entry points and loaders from the configuration

# Module federation (micro front end)

# References

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

<https://blog.ag-grid.com/webpack-tutorial-understanding-how-it-works/>

<https://medium.com/@cgcrutch18/commonjs-what-why-and-how-64ed9f31aa46>

<https://nodejs.org/docs/latest/api/modules.html>

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules>

<https://requirejs.org/docs/whyamd.html>

<https://stackoverflow.com/questions/24581873/what-exactly-is-hot-module-replacement-in-webpack>

<https://javascript.info/modules-dynamic-imports>

<https://github.com/ronami/minipack/blob/master/src/minipack.js>

<https://github.com/TheLarkInn/compare-webpack-target-bundles>

<https://createapp.dev/webpack>

<https://stackoverflow.com/questions/42523436/what-are-module-chunk-and-bundle-in-webpack>

<https://webpack.js.org/glossary/>

<https://medium.com/webpack/webpack-4-code-splitting-chunk-graph-and-the-splitchunks-optimization-be739a861366>