**WebPack**

* JavaScript is becoming more complex in size during the lifetime of an app. One of the main things WebPack can do is compile our JavaScript to one file or more like for ex: If we are prefer to call one JS script in one web page and another script in another web page.
* Compiling JS files into one single large file will prevent multiple server hits.
  + For ex: If our app uses multiple third party libraries like jQuery, tinymce, bootstrap, loadash etc.,

Instead of calling each one separately like this:

<script src="/assets/javascripts/jquery.js" /></script>

<script src="/assets/javascripts/tinymce.js" /></script>

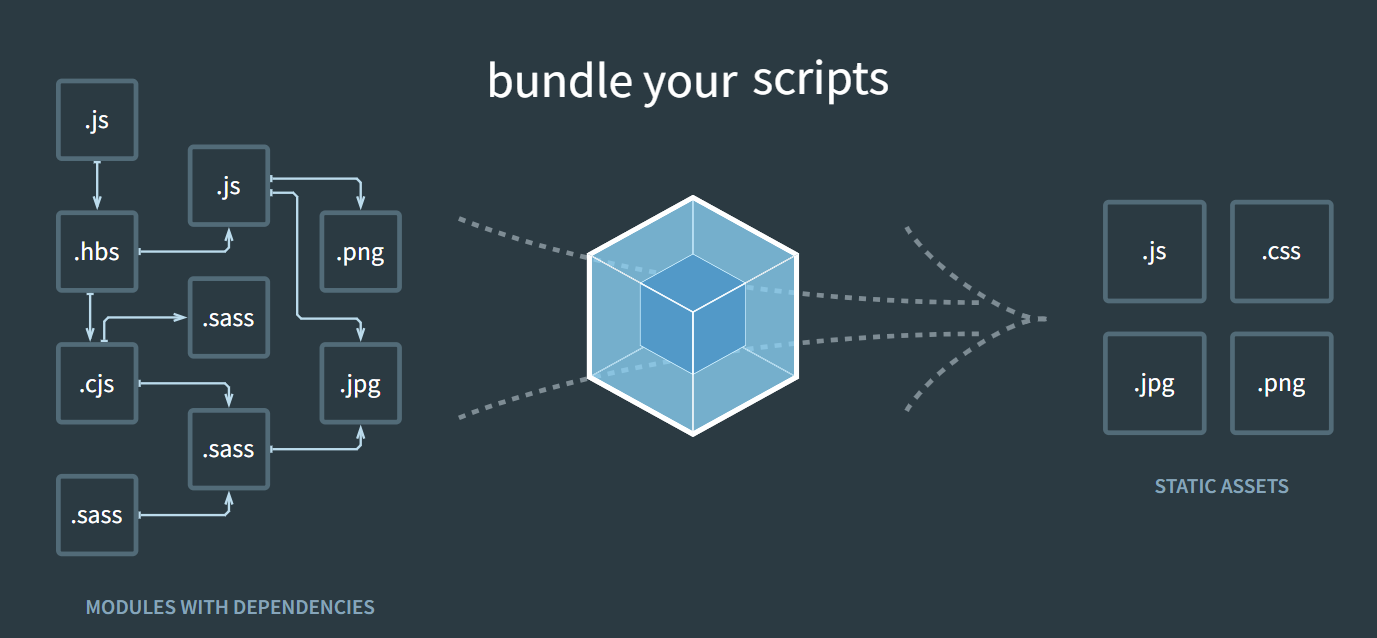
<script src="/assets/javascripts/bootstrap.js" /></script>

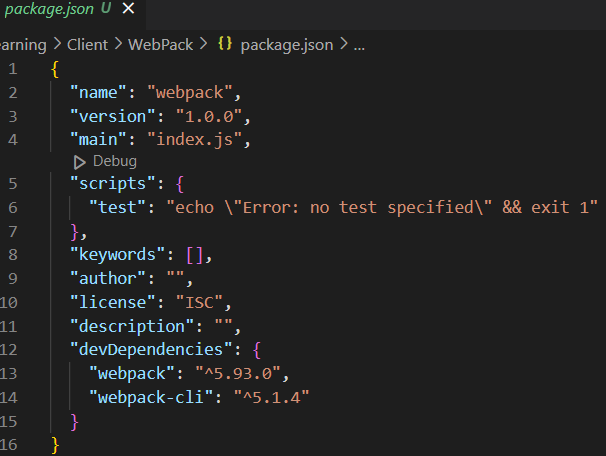
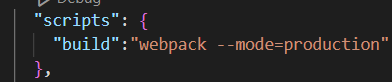
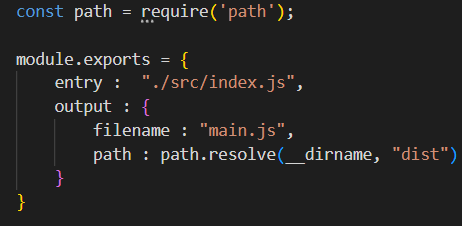
<script src="/assets/javascripts/loadash.js" /></script>

<script src="/assets/javascripts/some\_other\_library.js" /></script>

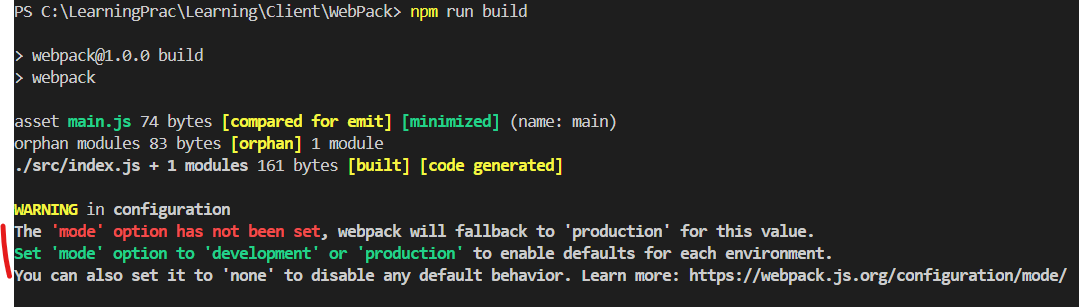
You can put all your third-party libraries in a file vendor.js and call it in your HTML with one line:

<script src="/assets/javascripts/vendor.js" /></script>



* npm install webpack webpack-cli –save-dev -🡪 This command installs webpack
* Since WebPack 4 there is no need for webpack.config.js file as Webpack 4 by default is a zero config which means that  Webpack expects that entry point for the app is index.js inside an *src* folder and then it will create the output in *dist/main.js* file for us without the need to create any config file.
* node\_modules/.bin/webpack 🡪 This command can be used to ask webpack to compile our js code.
* Instead, we can have a similar command in package.json like here :
* Webpack has zero config meaning that it expects us have our .js files in *src/* folder so that it can generate the file in *dist/main.js.* If we want to have zero config then it expects us to have this structure.
* But what if we want to configure things in our own way then we should have webpack.config.js as this will be default file webpack will be looking into.
*  Here it says that entry point is ‘index.js’ and output should be main.js
* The ‘Path’ should always be absolute path so the ‘Path’ module is going to resolve it for us.
* To have webpack in development mode we can simply add ‘mode:Development’ in the webpack.config.js file.

If mode is not set in package.json then we will end up getting below warning :



* With ‘watch=true’ setting we can make sure that we don’t need to run ‘npm run build’ , we will run ‘npm run build’ only for the first time and then from next time every time we save a js file webpack updates.
* **Loaders & Plugins:**
* **Loaders 🡪** These tell webpack on how to interpret, translates and transform our source code. They basically preprocess the files when we import them.

**Ex:** CoffeeScript to vanilla JavaScript, SCSS to CSS

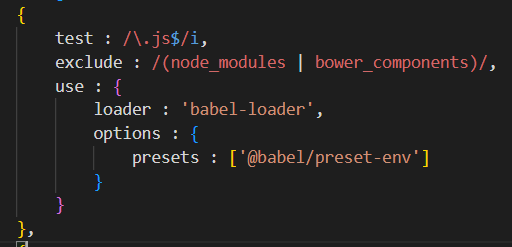
* **Plugins** 🡪 They can instantiated and can take arguments in order to perform many things or adding functionalities to compilation.

**Ex:**  new MiniCssExtractPlugin({ filename : ‘default.css’ })

The two steps we follow are :

1. Installing a loader/plugin
2. Call it with ‘require’ in the webpack.config.js and use it

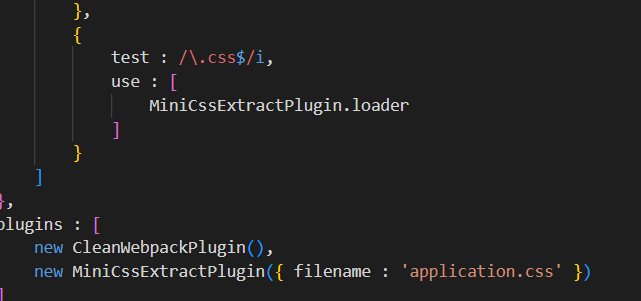
**Babel-loader**:

* This is used to convert (transpile) from one version of JavaScript to another version. For Ex: If we want to transpile from ES 6 version of JS(not supported by all browsers currently) to old version of JS (for older browsers) we will have to use BabelJS that transpiles from ES 6 to older version of JS.
* Webpack provides us babel-loader which can be used to solve the same problem
* 
  + /\.js$/i
    - \. matches the literal period (.) character
    - js matches the string “js”
    - $ asserts the position at the end of the string, ensuring that “js” is at the end of the file name
    - i 🡪 This flag makes the regex case-insensitive so that it will match “.js” even if “.JS” or “.Js” is present

**CSS Extraction** to separate file:

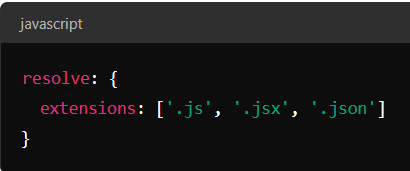
* If we have all the css into a separate file we can use **mini-css-extract-plugin**





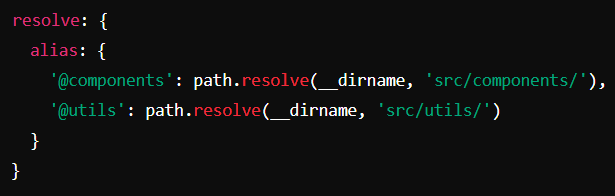
**Resolve**

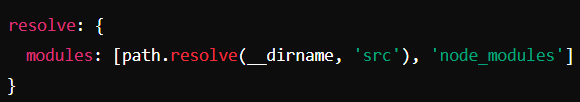
* This property tells us on how to resolve modules. It specifies the options for resolving module paths, aliases, extensions and other related things.
  + Extensions: Automatically resolve certain file extensions. This allows you to omit file extensions when importing modules.

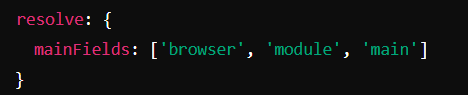


So, in this example webpack can automatically resolve imports for .js, .jsx, and .json files, so you can import like this: import App from './App' without needing to specify the file extension.

* Aliases: Create aliases for module paths to simplify imports and manage long or complex paths.



* modules : This explains us where Webpack should look for modules  This tells Webpack to first look for ‘src’ folder for packages before looking into ‘node\_modules’ folder.
* mainfields: Specifies which fields in package.json to look for resolving modules.



This means Webpack will first look for ‘browser’ field followed by ‘module’ followed by ‘main’ to determine which entry point to use.

* Plugins: Extends module resolution with custom plugins.