WEBPACK

What is webpack

- Webpack is a powerful open-source JavaScript module bundler primarily used in modern web development. It takes modules with dependencies and generates static assets representing those modules.
- webpack is a static module bundler for modern JavaScript applications. When webpack processes your application, it internally builds a dependency graph from one or more entry points and then combines every module your project needs into one or more bundles, which are static assets to serve your content from.

It can perform many operations

- Helps you bundle your resources.
- Watches for changes and re-runs the tasks.
- Can run the Babel transcompiler to ES5, allowing you to use the latest JavaScript features without worrying about browser support.
- Can convert inline images to data URIs.
- Allows you to use require() for CSS files.
- Can run a development web server.
- Can handle hot module replacement (HMR) – allows developer to update code in a running application without a full page reload, preserving app state and speeding up development.
- You can split the output files into multiple files to avoid having a huge JS file load on the first-page hit.

Webpack vs NPM

Feature	Webpack	NPM
Primary Function	Module bundler	Package manager
Installation Command	npm install --save-dev webpack	npm install <package-name>
Main Use	Bundles JavaScript files for browsers	Manages dependencies for Node.js
Compatibility	ES5-compliant browsers	Node.js environments
Advantages	Modular plugin system, loaders for file preprocessing	Package version management, wide repository of packages
Speed	Optimized bundling process	Yarn installs packages faster than NPM
Lock File	Not by default (can be configured)	package-lock.json (NPM), yarn.lock (Yarn)

Key Features of Webpack

- Module Bundling: Webpack takes various modules (JavaScript, CSS, images, etc.) and bundles them into a smaller set of files. This is especially useful for managing dependencies and optimizing load times.
- Code Splitting: It can split your code into smaller chunks, which can be loaded on demand or in parallel. This reduces the initial load time and improves performance.

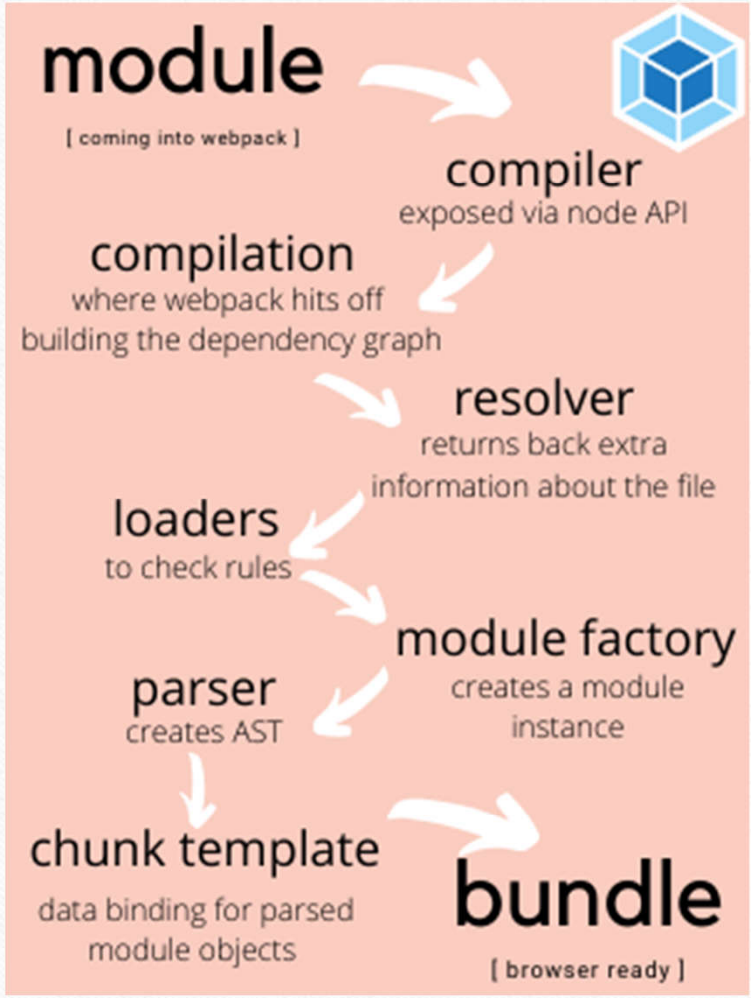
- Loaders: These are transformations that are applied to the source files of your project. For example, you can use loaders to transpile TypeScript to JavaScript, convert Sass to CSS, or load images as data URLs.

- Plugins: Plugins are used to perform a wider range of tasks like bundle optimization, asset management, and injection of environment variables. Webpack's plugin system is very powerful and can be customized to fit any need.

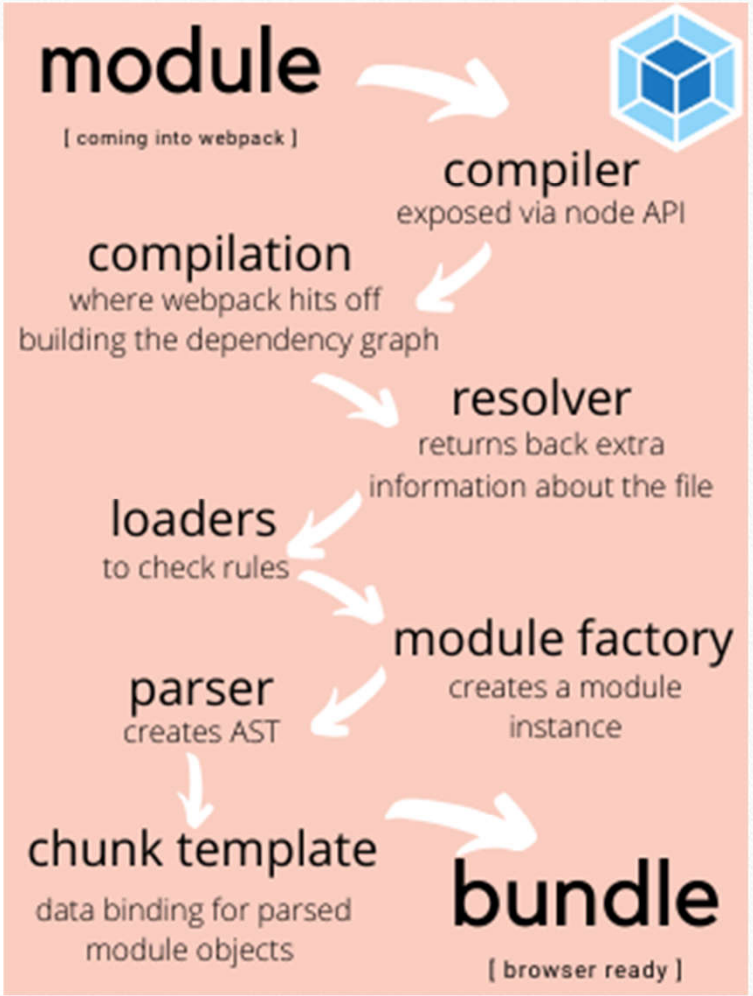
- Dev Server: Webpack Dev Server provides live reloading and other development features, making it easier to develop and test your applications locally.

Journey of a Module in Webpack

1. **Module (Entry Point)**
   - The starting point defined in the configuration file.
   - Webpack uses this entry module to begin building the dependency graph.
2. **Compiler**
   - Exposed via Node.js API.
   - Accepts the entry module and initiates the compilation process.
3. **Compilation**
   - Webpack constructs the dependency graph by traversing `import` or `require` statements.
   - Plugins may hook into this phase for custom processing.
4. **Resolver**
   - Determines the full path and additional information about each module.
   - Helps in locating modules based on configuration like `resolve.alias`, `extensions`, etc.
5. **Loaders**
   - Applied to transform files based on rules defined in `webpack.config.js`.
   - Example: Transpile ES6 with Babel or process `.scss` into CSS.



6. **Module Factory**
    - Responsible for creating module instances.
    - Incorporates transformed source and metadata.
7. **Parser**
    - Converts the module content into an Abstract Syntax Tree (AST).
    - Helps webpack understand dependencies and internal structure.
8. **Chunk Template**
    - Bundles the parsed module objects into code chunks.
    - Supports features like code splitting and lazy loading.
9. **Bundle**
    - Final browser-ready output.
    - Optimized and minified, ready to be served to users.



Core Concepts

- Entry
- Output
- Loaders
- Plugins
- Mode
- Browser Compatibility

1.Entry Point

- The entry point is the module where Webpack starts building the dependency graph. This is usually the main JavaScript file of the application.
- In this example, webpack will start from ./src/index.js and include all dependencies starting from this file.

```
// webpack.config.js
module.exports = {
  entry: './src/index.js',  // Entry point
  output: {
    filename: 'bundle.js',
    path: __dirname + '/dist',
  },
};
```

2. Output

- The output property tells Webpack where to emit the bundles it creates and how to name these files.

```
// webpack.config.js
const path = require('path');

module.exports = {
  entry: './src/index.js',
  output: {
    filename: 'bundle.js',  // Name of the output file
    path: path.resolve(__dirname, 'dist'),  // Output directory
  },
};
```

3.Loaders

- Loaders are transformations that are applied on the source files of your application. They allow us to preprocess files as we import or load them.
- Loaders allow webpack to process other types of files and convert them into valid modules that can be consumed by your application and added to the dependency graph.
- At a high level, loaders have two properties in your webpack configuration:
  - The test property identifies which file or files should be transformed.
  - The use property indicates which loader should be used to do the transforming.


module.exports = {
entry: './src/index.js',
output: {
filename: 'bundle.js',
path: __dirname + '/dist',
},
module: {
rules: [
{
test: /\.css$/, // Apply this rule to .css files
use: ['style-loader', 'css-loader'], // Use these loaders
},
{
test: /\.(png|jpg)$/, // Apply this rule to image files
use: ['file-loader'],
},
],
},
};

Commonly used Loaders

- babel-loader: Transpiles modern JavaScript to older versions for compatibility.
- css-loader: Resolves @import and url() in CSS files.
- style-loader: Injects CSS into the DOM.
- sass-loader: Compiles Sass/SCSS files to CSS.
- less-loader: Compiles Less files to CSS.
- ts-loader: Transpiles TypeScript to JavaScript.

Loaders

- file-loader: Emits files and returns their URLs.
- url-loader: Inlines files as base64 URLs if they are smaller than a specified limit.
- html-loader: Exports HTML as a string and can handle image sources.
- raw-loader: Loads file contents as a string.
- json-loader: Loads JSON files.
- vue-loader: Processes Vue.js single-file components.

Loaders

- eslint-loader: Lints JavaScript files using ESLint.
- babel-loader: Integrates Babel for JavaScript transpilation.
- image-webpack-loader: Optimizes image files.
- csv-loader: Loads CSV files and converts them to JSON.
- xml-loader: Loads XML files and converts them to JSON.
- graphql-tag/loader: Imports GraphQL files into JavaScript.

4.Plugins

- While loaders are used to transform certain types of modules, plugins can be leveraged to perform a wider range of tasks like bundle optimization, asset management and injection of environment variables.

```
const HtmlWebpackPlugin = require('html-webpack-plugin');
const webpack = require('webpack'); //to access built-in plugins

module.exports = {
  module: {
    rules: [{ test: /\.txt$/, use: 'raw-loader' }],
  },
  plugins: [new HtmlWebpackPlugin({ template: './src/index.html' })],
};
```

Plugin (continue).

- In order to use a plugin, you need to require() it and add it to the plugins array. Most plugins are customizable through options. Since you can use a plugin multiple times in a configuration for different purposes, you need to create an instance of it by calling it with the new operator.

Name	Description
BannerPlugin	Add a banner to the top of each generated chunk
ChunksWebpackPlugin	Create HTML files with entrypoints and chunks relations to serve your bundles
CommonsChunkPlugin	Extract common modules shared between chunks
CompressionWebpackPlugin	Prepare compressed versions of assets to serve them with Content-Encoding
ContextReplacementPlugin	Override the inferred context of a require expression
CopyWebpackPlugin	Copies individual files or entire directories to the build directory

DefinePlugin	Allow global constants configured at compile time
DllPlugin	Split bundles in order to drastically improve build time
EnvironmentPlugin	Shorthand for using the DefinePlugin on process.env keys
EslintWebpackPlugin	A ESLint (Static Code Analysis) plugin for webpack
HotModuleReplacementPlugin	Enable Hot Module Replacement (HMR)
HtmlWebpackPlugin	Easily create HTML files to serve your bundles
IgnorePlugin	Exclude certain modules from bundles
LimitChunkCountPlugin	Set min/max limits for chunking to better control chunking

MinChunkSizePlugin	Keep chunk size above the specified limit
MiniCssExtractPlugin	creates a CSS file per JS file which requires CSS
NoEmitOnErrorsPlugin	Skip the emitting phase when there are compilation errors
NormalModuleReplacementPlugin	Replace resource(s) that matches a regexp
NpmInstallWebpackPlugin	Auto-install missing dependencies during development
ProgressPlugin	Report compilation progress

ProvidePlugin	Use modules without having to use import/require
SourceMapDevToolPlugin	Enables a more fine grained control of source maps
EvalSourceMapDevToolPlugin	Enables a more fine grained control of eval source maps
SvgChunkWebpackPlugin	Generate SVG sprites optimized by SVGO based on your entry point dependencies
TerserPlugin	Uses Terser to minify the JS in your project

5. Mode

- By setting the mode parameter to either development, production or none, you can enable webpack's built-in optimizations that correspond to each environment. The default value is production.

```
// webpack.config.js
module.exports = {
  entry: './src/index.js',
  output: {
    filename: 'bundle.js',
    path: __dirname + '/dist',
  },
  mode: 'development',  // Set the mode to development
};
```

6. Browser Compatibility

- Webpack helps ensure that the code is compatible with the target browsers. It works with tools like Babel to transpile modern JavaScript into a version that can run in older browsers.

```
module: {
  rules: [
    {
      test: /\.js$/,
      exclude: /node_modules/,
      use: {
        loader: 'babel-loader',
        options: {
          presets: ['@babel/preset-env'],  // Transpile modern JS to ES5
        },
      },
    },
  ],
```

complete Webpack configuration file

that incorporates all these concepts:

```
// webpack.config.js
const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');

module.exports = {
  entry: './src/index.js',  // Entry point
  output: {
    filename: 'bundle.js',  // Output filename
    path: path.resolve(__dirname, 'dist'),  // Output directory
  },
  mode: 'development',  // Mode
  module: {
    rules: [
      {
        test: /\.css$/,  // CSS loader
        use: ['style-loader', 'css-loader'],
      },
```

```
    {
      test: /\.(png|jpg)$/,  // Image loader
      use: ['file-loader'],
    },
    {
      test: /\.js$/,  // Babel loader for JS files
      exclude: /node_modules/,
      use: {
        loader: 'babel-loader',
        options: {
          presets: ['@babel/preset-env'],
        },
      },
    },
  ],
},
```

```
  plugins: [
    new HtmlWebpackPlugin({
      template: './src/index.html',  // HTML template
    }),
  ],
};
```

This configuration will take index.js as the entry point, process CSS and image files, transpile JavaScript for browser compatibility, and generate an HTML file with the bundled scripts.

Dependency Graph

- Any time one file depends on another, webpack treats this as a dependency. This allows webpack to take non-code assets, such as images or web fonts, and also provide them as dependencies for your application.
- When webpack processes your application, it starts from a list of modules defined on the command line or in its configuration file. Starting from these entry points, webpack recursively builds a dependency graph that includes every module your application needs, then bundles all of those modules into a small number of bundles - often, only one - to be loaded by the browser.

comparison of web development workflows with and without Webpack

Aspect	Without Webpack	With Webpack
File Management	Manual inclusion of separate files	Single entry point with automatic bundling
Dependency Management	Manual order management	Automated dependency graph
Transpiling/Preprocessing	Separate tools for Babel, Sass, etc.	Loaders for transpiling and preprocessing
Code Splitting	No code splitting	Automatic code splitting
Initial Load Time	Slower, multiple HTTP requests	Faster, fewer HTTP requests
Optimizations	Manual optimizations	Automatic minification, tree shaking, etc.
Development Server	Manual setup	Webpack Dev Server with HMR
File Inclusion in HTML	Manual script/style tags in HTML	HtmlWebpackPlugin for automatic injection
Error Prone	More error-prone	Less error-prone, automated processes
Build Process	More time-consuming	Efficient, streamlined build process

Benefits of Using Webpack

- **Efficiency**: It helps in optimizing the assets of your application, resulting in faster load times.
- **Modularity**: Encourages a modular approach to development, where each part of the application can be developed, tested, and maintained independently.
- **Ecosystem**: A vast ecosystem of loaders and plugins that can be used to tailor the build process to specific needs.
- **Community Support**: Strong community support with extensive documentation and a wealth of tutorials and resources.

Working with Webpack

1. Create a folder name basic-webpack-code

2. Inside it create the package.json by running the command npm init -y

3. Install Webpack **npm i webpack webpack-cli** provides a flexible set of commands for developers to increase speed when setting up a custom webpack project.

4. Create the Webpack configuration file **create the webpack.config.js** inside the root folder

5. Once created add the following code inside webpack.config.js

webpack.config.js > ...

```
const path = require("path");
module.exports = {
  entry: "./src/index.js",
  output: {
    path: path.resolve(__dirname, "dist"),
    filename: "bundle.js",
  },
};
```

The entry point is the file Webpack will run through to create our bundle.

The output which contains sets the name of our final bundle file and where to place it

6. Now let's create our entry file index.js in the src folder.

mkdir src

touch src/index.js

7. Then add some simple code inside index.js.

src > JS index.js > ...

```
1  window.onload = function () {
2    console.log("Hello World");
3  };
```

8. Now add this “bundle: “webpack –config ./webpack.config.js” in the scripts section and run **npm run bundle** in the terminal. As an outcome, we will see a new folder, named **dist**, which contains our bundle.js.

bundle.js would be a single file that contains all our compressed javascript.

```
"scripts": {
  "bundle": "webpack --config ./webpack.config.js "
},
```

— config is very useful in case you want to use different config files for different scripts.

9. Let’s Create an Html file in the root folder and require our bundle.js in there

index.html > ...

```html
<!DOCTYPE html>
<html lang="en" dir="ltr">
  <head>
    <meta charset="utf-8" />
    <title></title>
  </head>
  <body>
    <h1>Well done learning more about Webpack</h1>
    <script type="text/javascript" src="./dist/bundle.js"></script>
  </body>
</html>
```

10. Let's install webpack-dev-server **npm i webpack-dev-server -D** webpack-dev-server is going to serve our bundle.js using the index.html file placed in the root.

In order to configure it we have to go to the webpack.config.js and add in the module object the devServer key and the basic following properties:

webpack.config.js > <unknown>

```
const path = require("path");

module.exports = {
  entry: "./src/index.js",
  mode: "development",
  output: {
    path: path.resolve(__dirname, "dist"),
    filename: "bundle.js",
  },
  devServer: {
    static: {
      directory: path.join(__dirname, "./dist"),
    },
    compress: true,
    port: 8050,
  },
};
```

port = specifies the port where to run the server directory = specify the path where the webpack dev server will find all the necessary files (index.html
and bundle .js).

11. By default, the webpack-dev-server will be looking for our index.html inside the dist folder and nowhere else. To make it work we need Html Webpack Plugin and add plugin configuration inside webpack.config.js.

npm i html-webpack-plugin -D

webpack.config.js > ...

```
const path = require("path");
var HtmlWebpackPlugin = require("html-webpack-plugin");
module.exports = {
  entry: "./src/index.js",
  mode: "development",
  output: {
    path: path.resolve(__dirname, "dist"),
    filename: "bundle.js",
  },
  devServer: {
    static: {
      directory: path.join(__dirname, "./dist"),
    },
    compress: true,
    port: 8050,
  },
  plugins: [
    new HtmlWebpackPlugin({
      filename: "index.html",
      template: "./index.html",
    }),
  ],
};
```

12. Before trying it out, add some text changes in the HTML file and REMOVE the script we used to import the bundle.js with.

Because our HTML plugin is going to take care of that as well! Is going to automatically add the import of the file specified in the entry point key in our webpack.config.js file.Change script bundle to build and run npm run build, after this run npm start, it will work in localhost:8050.

```
"scripts": {
  "build": "webpack --config ./webpack.config.js ",
  "start": "webpack-dev-server --config ./webpack.config.js"
},
```

13. Now, we have a running application ,let's check this mode property also , Add below scripts with **mode=development** and **mode=production** in package.json file.

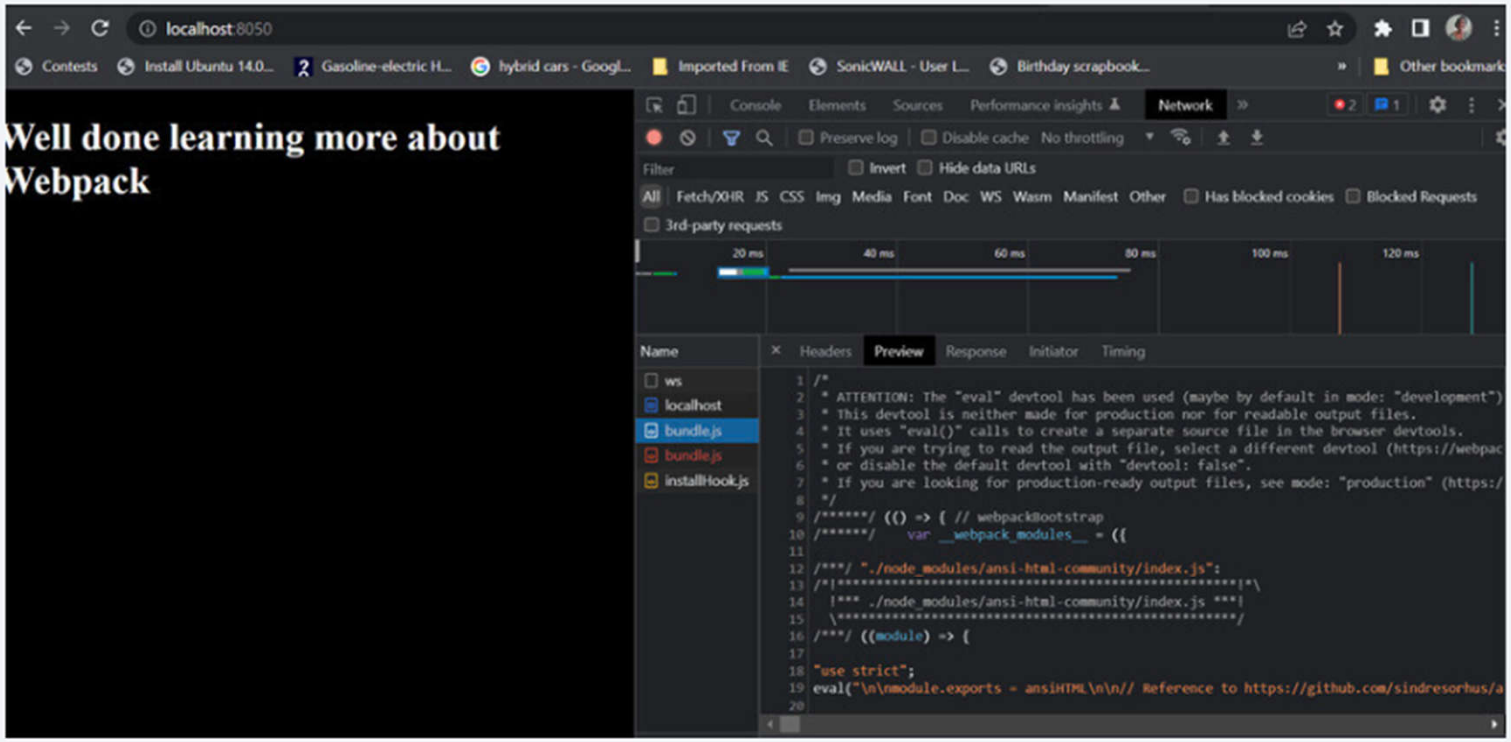
"start-dev": "webpack-dev-server --mode=development --config ./webpack.config.js --open",

"start-prod": "webpack-dev-server --mode=production --config ./webpack.config.js --open"

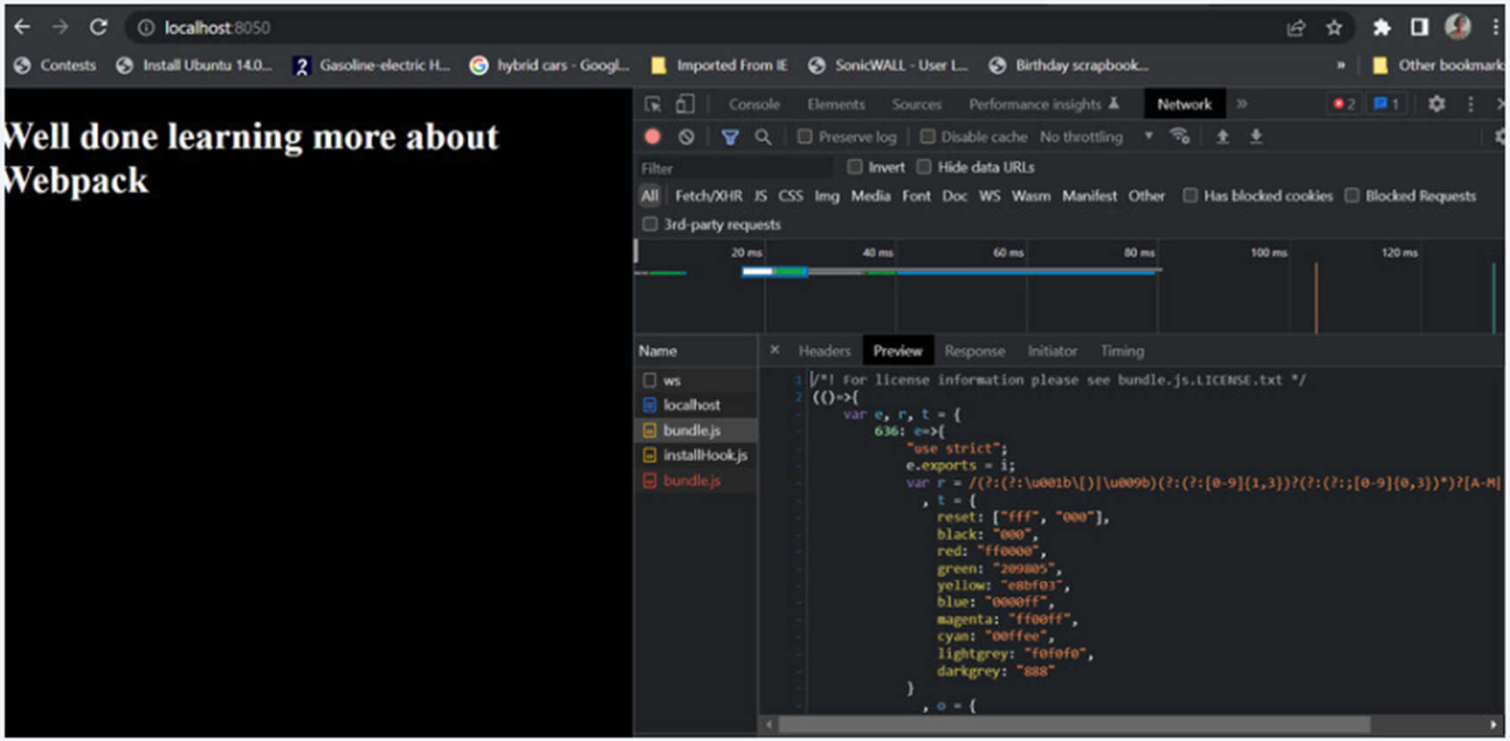
14. Convert module.exports into function and pass two arguments env, argv. We can access mode in webpack by **argv.mode**.

```
const path = require("path");
var HtmlWebpackPlugin = require("html-webpack-plugin");
module.exports = (env, argv) => {
  return {
    entry: "./src/index.js",
    mode: argv.mode,
    devtool: argv.mode === "production" ? "source-map" : "eval",
    output: {
      path: path.resolve(__dirname, "dist"),
      filename: "bundle.js",
    },
    devServer: {
      static: {
        directory: path.join(__dirname, "./dist"),
      },
      compress: true,
      port: 8050,
    },
```

15. Now if you run **npm run start-dev** you will see bundle.js size as **273 KB** and if you click on bundle.js , you will see the generated bundle contains a lot of comments.



16. If you run **npm run start-prod** bundle.js size will be **124 KB .** In production mode Webpack does not insert anything extra in production mode, just the code that is required.



Now let’s learn about loader through hands-on

1. Inside the src folder, create another folder called style and inside it, one file called main.scss

mkdir src/styles

touch src/styles/main.scss

2. Import your SCSS file in the index.js file placed in the src folder

import ‘./styles/main.scss’

src > styles > main.scss > ...

```scss
body {
  background: black;
  color: white;
  margin: 0;
  padding: 0;
}
```

3. If you now try to build the app Webpack is gonna tell you exactly what you need.

   You may need an appropriate loader to handle this file type.

   You need a Loader. This is because by default Webpack is only capable of bundling js files and in this case, we need something to first of all convert scss to css and then css to js!

4. The loader we will use is sass-loader.

   npm i style-loader css-loader sass-loader node-sass -D

   Add this module section in webpack.config.js

   Now do npm run build and npm run start, it will work.

```
module: {
  rules: [
    {
      test: /\.scss$/,
      use: [
        "style-loader", // creates style nodes from JS strings
        "css-loader", // translates CSS into CommonJS
        "sass-loader", // compiles Sass to CSS, using Node Sass by default
      ],
    },
  ],
},
};
```

5. Now create the default add function in the src folder and add the below code , import this as add-in index.js and call add(5,10) in index.js .

src > JS add.js > default

```
1  export default (a, b) => {
2  return a + b;
3  };
```

6. Now if you build and do npm start you will see an error on the browser, it is because webpack only understands ES5 code. To compile our ES6+ code to ES5 we need a babel-loader.

```
module: {
  rules: [
    {
      test: /\.js$/,
      exclude: /node_modules/,
      loader: "babel-loader",
      options: {
        presets: ["env"],
      },
    },
    {
      test: /\.scss$/,
      use: [
        "style-loader", // creates style nodes from JS strings
        "css-loader", // translates CSS into CommonJS
        "sass-loader", // compiles Sass to CSS, using Node Sass by default
      ],
    },
  ],
},
```

Now again npm run build and npm run start, it will work.