Lesson - 29

Module Bundler. Webpack

PARCEL

Blazing fast, zero configuration web application bundler

webpack

MODULE BUNDLER

rollup.js

SkillUp by Dobrea Vladislav

Lesson Plan

- HW Review
- Webpack
- Loaders
- Dev server
- Build

What is a module bundler?

Module bundlers are tools frontend developers used to bundle **JavaScript modules** into a single JavaScript files that can be executed in the browser.

Examples of modern module bundlers *(in no particular order)* are: **webpack**, **rollup**, **fusebox**, **parcel**, etc.

Module bundler is required because:

- It helps optimize and eliminate un-used code
- Browser does not support module system, **although this is not entirely true nowadays**
- It helps you manage the dependency relationship of your code, it will load modules in dependency order for you.
- It helps you to load your assets in dependency order, image asset, css asset, etc.

What is webpack?

Webpack is a module bundler. It takes disparate dependencies, creates modules for them and bundles the entire network up into manageable output files. This is especially useful for **Single Page Applications** (SPAs), which is the defacto standard for Web Applications today.

However, Webpack is more than just a module bundler. With the help of loaders and plugins, it can transform, minify and optimize all types of files before serving them as one bundle to the browser. It takes in various assets, such as JavaScript, CSS, Fonts, Images, and HTML, and then transforms these assets into a format that's convenient to consume through a browser. The true power of Webpack is the sum of its parts.

Webpack core concepts

Webpack builds a dependency graph when it processes your application. It starts from a list of modules defined in its config file (webpack.config.js) and recursively builds a dependency graph that includes every module your application needs, then packages all of those modules into a small number of bundles to be loaded by the browser.

There are four core concepts you need to grasp to understand how Webpack functions:

- Entry
- Output
- Loaders
- Plugins

Webpack entry

Every Webpack setup has one or more *entry points.* The entry point tells Webpack where to start building its dependency graph from. Webpack starts processing the module at the entry point and roams around the application source code to look for other modules that depend on the entry module. Every direct or indirect dependency is captured, processed and outputted into a bundle(s).

webpack.config.js

```
const config = {
  entry: './app/prosper.js'
};

module.exports = config;
```

webpack.config.js - separate entries

```
const config = {
  entry: {
    app: './src/app.js',
    vendors: './src/vendors.js'
  }
};

module.exports = config;
```

Webpack output

Only one output point can be specified in a Webpack setup. The *output* config property tells Webpack where to place the bundle(s) it creates and how to name them. It is as simple as specifying the output property in the config file like so:

```
const config = {
    entry: './app/prosper.js',
    output: {
        path: '/unicodeveloper/project/public/dist',
        filename: 'app.bundle.js'
    }
};

module.exports = config;
```

output.filename - The name of the *bundle* webpack produces. output.path - The directory to write app.bundle.js to.

Webpack loaders

Loaders are like transformers. With loaders, Webpack can process any type of file, not just JavaScript files. Loaders transform these files into modules that can be included in the app's dependency graph and bundle. Check out the example below:

```
const config = {
  entry: './app/prosper.js',
  output: {
    path: '/unicodeveloper/project/public/dist',
    filename: 'app.bundle.js'
  },
  module: {
    rules: [
      { test: /\.(html)$/, use: 'html-loader' }
    ]
  }
};

module.exports = config;
```

In the code above, the html-loader processes HTML files and exports them as strings. There are several loaders such as css-loader, less-loader, coffee-loader and many more.

Webpack plugins

Earlier in the post, I mentioned that loaders are like transformers. Plugins are *super-man-like* in their operations. They can do a lot more tasks than loaders. In fact, just anything ranging from deleting files, to backing up files on services like Cloudinary, to copying files, etc. Check out the example below:

```
const CompressionWebpackPlugin = require('compression-webpack-plugin');
const webpack = require('webpack');
const path = require('path');

const config = {
  entry: './app/prosper.js',
  output: {
    path: '/unicodeveloper/project/public/dist',
    filename: 'app.bundle.js'
  },
  module: {
    rules: [
      { test: /\.(html)$/, use: 'html-loader' }
    ]
  },
  plugins: [
    new CompressionWebpackPlugin({test: /\.js/})
  ]
};

module.exports = config;
```

Loaders for CSS

A quick breakdown of what each of these plugins and loaders aims to accomplish.

- sass-loader - Loads a SASS/SCSS file and compiles it to CSS. It requires node-sass to work.
- node-sass - This libarary allows you to natively compile .scss files to css at incredible speed and automatically via a connect middleware.
- extract-text-webpack-plugin - Extract text from a bundle, or bundles, into a separate file.
- css-loader - The css-loader interprets @import and url() like import/require() and resolves them.
- style-loader - Add CSS to the DOM.

Loaders for CSS

```
const path = require('path');
const CopyWebpackPlugin = require('copy-webpack-plugin');
const MiniCssExtractPlugin = require('mini-css-extract-plugin');

module.exports = {
    entry: './src/js/index.js',
    output: {
       path: path.resolve(__dirname, 'build'),
       filename: 'normal.js'
    },
    module: {
        rules: [
            {
                test: /\.(sa|sc|c)ss$/,
                use: [
                  {
                    loader: MiniCssExtractPlugin.loader,
                  },
                  'css-loader',
                  'sass-loader',
                ],
              },
        ]
      },
      plugins: [new MiniCssExtractPlugin({
        filename: 'styles.css'
      })],
};
```

Home work

1. Create an empty project and then follow instruction bellow:

Setup a basic webpack configuration (in root folder, create a file named webpack.config.js:

- You need to have 1 entry file (index.js), it should be in folder named “js”
- Your output should be in build folder named bundle.js

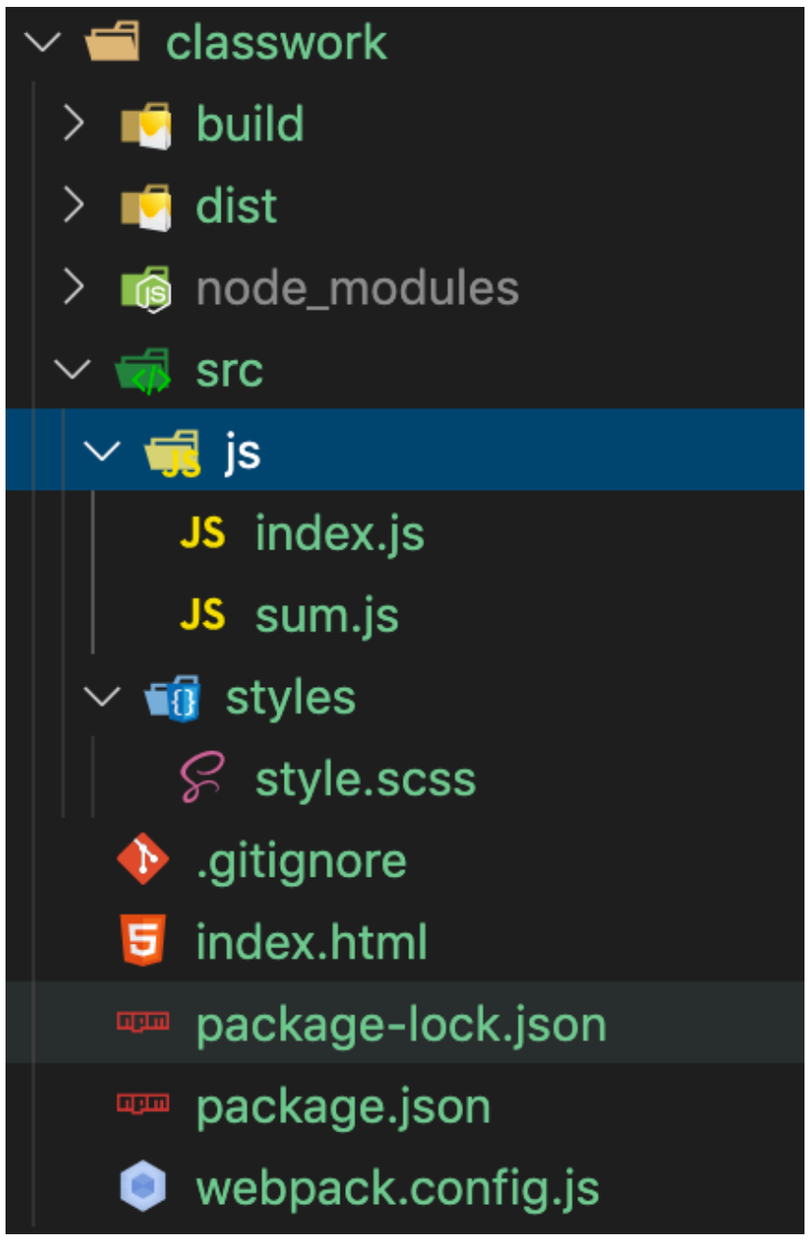
CSS:

- You need to have a CSS file extracted in build folder and named styles.css
- Your CSS should be written in SCSS, and be inside “src” in “styles” folder (check next page for folder structure)
- For css webpack config, use previous slide(Loaders for CSS).

- After your build is ready, you can now use your CSS and JS files in index.html, “./build/bundle.js” and “./build/styles.css”

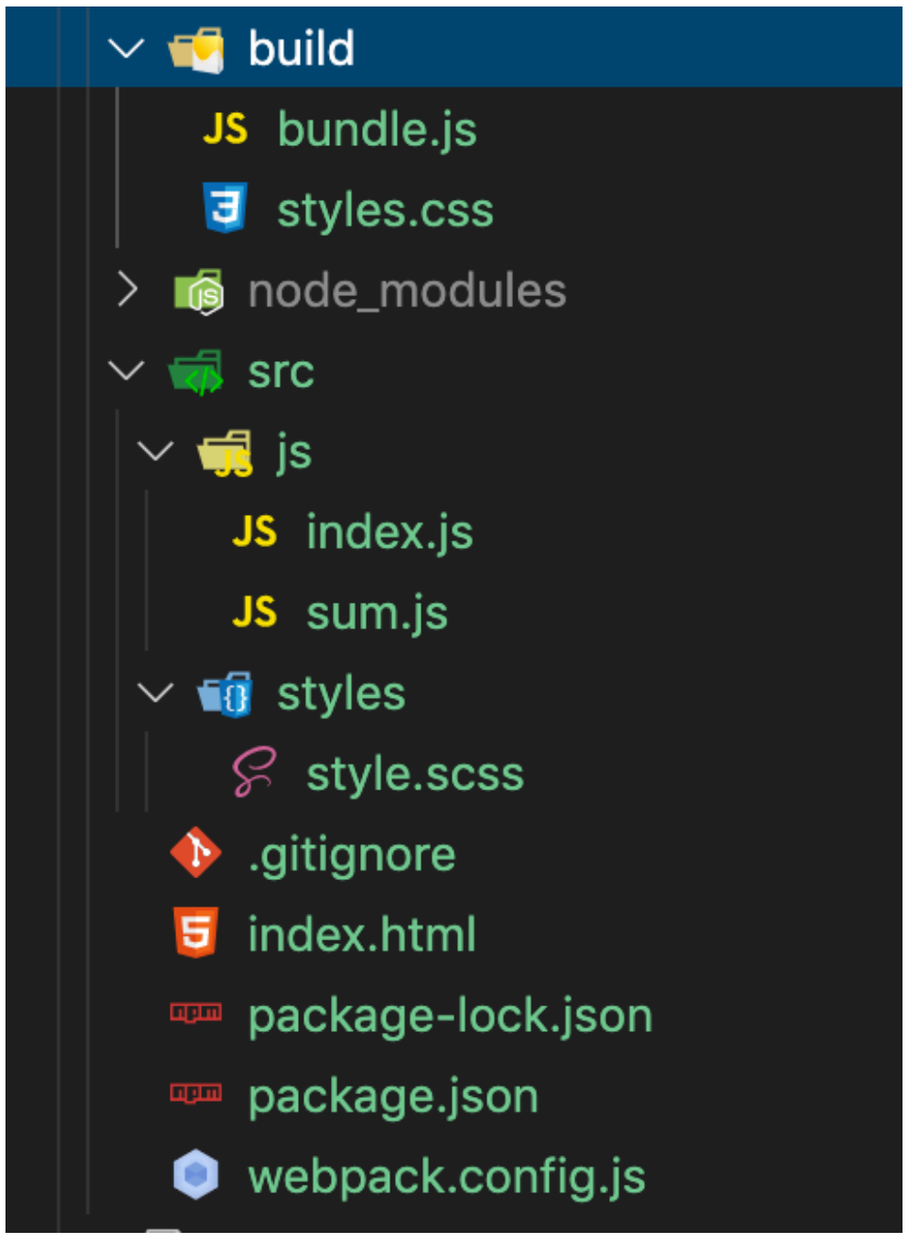
Home work

Project structure



Home work

Result expected



Learning Resources

1. **Webpack:**
    1. https://auth0.com/blog/webpack-a-gentle-introduction/
    2. https://medium.com/ag-grid/webpack-tutorial-understanding-how-it-works-f73dfa164f01
    3. https://www.youtube.com/watch?v=TzdEpgONurw
    4. **Official Documentation: https://webpack.js.org/**