Extra Notes - Webpack - Working with Asset Management

- Dr Nick Hayward

A quick outline to using Webpack bundler with asset management in web applications.

Contents

- Intro
- Webpack bundling of assets
- CSS assets
  - loading CSS
  - Webpack and CSS
  - minimising CSS
  - add multiple CSS files
- Image assets
  - import images
  - image object
  - image and CSS
- Font assets
- Data assets
  - add some data
- Global to Component assets
- References

Intro Webpack is a utility for bundling modules and project code for distribution release of browser compatible apps.

Getting started details may be found at the following URL,

- Webpack - Getting Started

Webpack may also be used to manage application assets as part of the build process for a given production.

Webpack bundling of assets Webpack also includes bundling of any file type, beyond JavaScript, as long as there is a supported loader.

So, as we find with JavaScript bundling, we may also gain from benefits such as explicit dependency support &c. as we build a web application.

CSS assets We can start by updating our app's structure, modifying the required JavaScript file in the app's `index.html`

```
<script src="./bundle.js"></script>
```

and then updating the `webpack.config.js` file as well

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

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

loading CSS To use CSS with Webpack, we need to import a CSS file from within a JavaScript module.

We need to install the `style-loader` and `css-loader` modules from NPM,

```
npm install style-loader css-loader --save-dev
```

To load this type of file, we can add `style-loader` and `css-loader` to our `webpack.config.js` file,

```
...
    module: {
        rules: [
            {
                test: /\.css$/,
                use: [
                    'style-loader',
                    'css-loader'
                ]
            }
        ]
    }
...
```

The above `test` rule uses a regular expression to check for compatible file types, in this example simply CSS files.

Any file with a `.css` file ending will be served by `style-loader` and `css-loader` .

Webpack and CSS As a specified CSS file is imported by Webpack, a `style` tag will be inserted into the `head` of the HTML file for rendering.

Any required CSS files need to be explicitly imported in the `index.js` file for the app, e.g.

```
import './style.css';
```

We may use the imported CSS directly in the JavaScript of an app, or simply style HTML as usual with the imported stylesheet.

minimising CSS We may also use Webpack to minimise CSS files for production usage in an app.

We need to install the following modules from NPM,

```
npm install terser-webpack-plugin --save-dev
npm install mini-css-extract-plugin --save-dev
npm install optimize-css-assets-webpack-plugin --save-dev
```

and then require them in the `webpack.config.js` file

```
/* css minimising */
const TerserPlugin = require("terser-webpack-plugin");
const MiniCssExtractPlugin = require("mini-css-extract-plugin");
const OptimizeCSSAssetsPlugin = require("optimize-css-assets-webpack-plugin");
```

We may then update this config file further to uglify and minimise our CSS files, e.g.

```
module.exports = {
    mode: 'production',
  entry: './src/index.js',
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist')
  },
    optimization: {
    minimizer: [
      new TerserPlugin(),
      new OptimizeCSSAssetsPlugin({})
    ]
  },
  plugins: [
    new MiniCssExtractPlugin({
      filename: "[name].css",
      chunkFilename: "[id].css"
    })
  ],
    module: {
        rules: [
            {
                test: /\.css$/,
                use: [
                    // 'style-loader',
                    MiniCssExtractPlugin.loader,
                    'css-loader'
                ]
            }
        ]
    }
};
```

If we then run the usual `npm run build` command, we get a `main.css` in the `dist` directory for our app.

We may reference this minimised CSS file in our as usual,

```
<link rel="stylesheet" type="text/css" href="./main.css">
```

add multiple CSS files We may import many additional single CSS files in the `index.js` file for the app.

Each file will be minimised to the single `main.css` file, which may be loaded as usual in the app's `index.html` file.

Image assets Using an additional loader, the `file-loader` , we may also include support for images within our Webpack based project.

Install with the following NPM command,

```
npm install file-loader --save-dev
```

We may then update our app's `webpack-config.js` file to test for image file names, and then specify the module to use.

e.g. update the `rules` array with the following object for images

```
...
{
    test: /\.(png|svg|jpg|gif)$/,
    use: [
        'file-loader'
    ]
}
...
```

We may also choose to limit the supported image file types by simply restricting those specified in the RegExp for the test.

n.b. `file-loader` and `url-loader` will take any file passed, and output it to the app's defined build directory. e.g. images, fonts…

import images With image support added to the project, we may import any required images in the JavaScript.

These images will then be processed, and placed in the `output` directory for the project. e.g.

```
import logo from './logo.png';
```

The image is then available for use within the JavaScript logic of the application.

image object We may simply create an image object, and then set the imported image as the value of the `src` attribute on the image element.

e.g.

```
const logoImage = new Image(78, 78);
logoImage.src = logo;// logoImage.setAttribute('src', logo);

logoImage.classList.add('logo');

document.getElementById('logo').append(logoImage);
```

In this example, we create a new image object, and set the following,

- height and width in pixels
- value of the `src` attribute
- add a new class to the `classList` property
- append image to the element with ID `logo`

image and CSS We may also import an use an image with our app's CSS.

e.g.

```
#logo {
    background: url('./tree.png');
}
```

Font assets We may also use the `file-loader` to include custom font files in app's build directory.

We need to update our app's `webpack.config.js` file,

```
...
{
    test: /\.(ttf|otf|eot|woff|woff2)$/,
    use: [
        'file-loader'
    ]
}
...
```

This test will simply check for supported font files in the `src` directory, and then import them to the app's build directory.

e.g. MyriadPro-Regular.ttf

We may then define this local font for use in CSS, e.g.

```
@font-face {
    font-family: 'Spire';
    src: url('./MyriadPro-Regular.ttf') format('ttf');
    font-style: normal;
}
```

The imported custom font file may then be referenced using the standard `font-family` property for a given ruleset in CSS.

Data assets We may also import various other file formats, including JSON, CSV, TSV, and XML.

For example, if we wanted to add support for importing CSV and XML, we can install the applicable loaders from NPM.

e.g.

```
{
    test: /\.(csv|csv)$/,
        use: [
            'csv-loader'
   ]
{
  test: /\.xml$/,
      use: [
     'xml-loader'
    ]
}
```

Add some data After configuring Webpack to process data files, we may add any required CSV, XML, &c. files to the app's `src` directory.

As the file is imported, each loader will return parsed JSON for use in the app's JS logic.

e.g.

```
import sitesData from './sites.xml'
```

We may then use this new JSON object in our app as usual.

Global to Component assets Webpack usage and integration also offers a useful benefit for project structure and asset management.

By referencing assets relative to a given component, as shown above, we may now group logic and assets together within a project structure.

Instead of a global assets directory, we may add images to their required components, and call local data files in the correct component context.

Another benefit is abstraction of component usage to permit easier code reuse and sharing.

e.g. a component might require the following directory structure

```
|-- components
|   |-- spire-component
|   |   |__ index.js
|   |   |__ spire.js
|   |   |__ style.css
|   |   |__ logo.png
```

The `spire-component` may now be processed with the required file dependencies without querying a separate assets directory, perhaps at the root of the app.

References

- Webpack - Getting Started