Set up a (React) project from scratch

written by Hong Le Nguyen 08.2024

We have to do some tasks

1)

Initialize an empty npm project --

Before you continue, make sure you have Node.js and npm already installed.

```
node -v ,  npm -v
```

create a new folder and initialize an empty npm project

```
cd hongBoilerplate4new
npm init -y
```

This will create a package.json file in the folder that describes an empty package.

2)

Add some example code to test the project –

Install React and some packages

```
npm install react react-dom
npm install react-router-dom axios uuid lodash
```

project structure

```
hongBoilerplate4new  // root directory
  build
  public
    index.html
  src
    index.js
    App.jsx

// public/index.html
<!DOCTYPE html>
<html lang="en">
 <head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1" />
  <title>Hello World</title>
 </head>
 <body>
  <div id="root"></div>
 </body>
</html>

// src/App.jsx
const App = () => (
  <h1>Hello World!</h1>
)
export default App;
```

```
// src/index.js
import ReactDOM from 'react-dom/client';
import App from './App.jsx';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <App />
);
```

3)
Install the webpack and tools

Webpack and Plugins , Loaders –

Webpack
is a bundler that uses a dependency graph to bundle code and assets into a 'bundle' which we can then deploy. The webpack-cli is a command line interface to use Webpack.

Plugins –
webpack can be extended by plugins. Webpack provides many such plugins out of the box.

html-webpack-plugin
this plugin automatically creates an HTML file with the reference of bundles created by Webpack.

clean-webpack-plugin
ensures that no old data gets left in the build folder after each build.

Loaders --
webpack uses loaders to preprocess files as you import them.

babel-loader
loads .js and .jsx files through Babel to transpile them before webpack bundles them.

Babel --
babel-loader needs the actual Babel package (@babel/core) in order to work.
Babel works by using 'plugins' to it's core package to transform and compile the code you give it.

preset-env
is such a plugin that compiles down ES6+ to ES5 standards-compatible JavaScript.

preset-react
is similarly another plugin that allows Babel to recognize and compile React code.

You can install them as dev-dependencies like so

Webpack and it's related plugins, loaders

```
npm install --save-dev webpack webpack-cli html-webpack-plugin clean-webpack-plugin babel-loader
```

Babel and it's related packages

```
npm install --save-dev @babel/core @babel/preset-env @babel/preset-react
```

4)
the webpack configuration ---

Webpack can be configured by placing a file named webpack.config.js in the project folder.
Place the following code inside it

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

module.exports = {

  entry: {
    main: path.resolve(__dirname, './src/index.js')
  },

  output: {
    path: path.resolve(__dirname, 'build'),
    filename: 'main.js'
  },

  plugins: [
    new HtmlWebpackPlugin({
      template: path.resolve(__dirname, './public/index.html')
    }),
    new CleanWebpackPlugin(),
  ],

  module: {
    rules: [
      {
        test: /\.(js|jsx)$/,
        exclude: /node_modules/,
        use: {
          loader: 'babel-loader',
          options: {
            presets: [
              ['@babel/preset-env', { targets: 'defaults' }],
              ['@babel/preset-react', { runtime: 'automatic' }]
            ]
          }
        }
      }
    ]
  },

}
```

This config instructs webpack to

Find all JS files and React components by recursively going through the imports of each file starting from index.js (entry)
Bundle all of the above and output it to a file named main.js in a folder named build (output)

Add a script tag referencing the outputted main.js file in the index.html file and place that in the build (HtmlWebpackPlugin)
clear out old files gets left in the build folder (CleanWebpackPlugin)
Use babel to process all of the .js and .jsx files it finds imported (module, rules)

We also pass some additional *options* into babel-loader to pass to babel.
Specifically, we ask it to use the @babel/preset-env and @babel/preset-react presets.

we ask @babel/preset-env to target browsers that are used by over 0.5% of world wide users, including the last 2 versions of all major browsers, by specifying "defaults" as the target.

@babel/preset-react handles converting JSX syntax into JS. The specifying { "runtime" : "automatic" } makes React object globally available.

5)
run production build --

We can now run the webpack command : *npx webpack --mode production*
to generate a production build. We pass production in the mode parameter so that it enables extra error checking for the build process and minification for the outputted code.

Add command to npm script --
It is more convenient to save this command as a npm script so that it can be run simply by using npm run build.
Add the following statement to the scripts object inside package.json

```
// package.json
{
  ...
  "scripts": {
    "build": "webpack --mode production"
  }
}
```

Now when you run : *npm run build*
a production build will be created and placed in the build folder.

6)
Adding support for CSS files ---

It also convenient to split the CSS code and store it alongside the React components as multiple .css or .scss files.

Setting up CSS with webpack –

Webpack need the help of plugins and loaders to process .css or .scss files. At first, let's install some dev dependencies.

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

The definition says css-loader interprets @import and url() like import/require() and will resolve them. What do we mean by this?
css-loader will take CSS from all the CSS files referenced in application, put it into a string and then ..
style-loader will take this string and will embed it in the style tag in index.html.

Now let's add the configuration in webpack.config.js

```
// webpack.config.js
…
module: {
      rules: [
        {
          test: /\.(css)$/,
          use: ['style-loader', 'css-loader']
        }
      ]
  }
```

Note: The order in which webpack apply loaders is *from last to first*, so as said earlier the css-loader will generate the output string which will be used by the style-loader.

Setting up Sass with webpack --

you need to install some packages to set up Sass with webpack.

```
npm install --save-dev sass-loader sass
```

sass-loader loads a Sass/SCSS file and compiles it to CSS.

Now let's update the webpack.config.js

```
// webpack.config.js
…
module: {
  rules: [
    …
    {
      test: /\.(css|scss)$/,
      use: ['style-loader', 'css-loader', 'sass-loader']
    }
```

We just need to add the sass-loader ahead of css-loader, so now first, the .scss file compiles back to CSS and after that process remains the same as explained above.

Configuration for your production environment –

You'll need a different configuration for using CSS. At first, install mini-css-extract-plugin

```
npm install --save-dev mini-css-extract-plugin
```

then replace the style-loader with MiniCssExtractPlugin.loader in configuration. MiniCssExtractPlugin extracts CSS and create a separate CSS file.

```
// webpack.config.js
const MiniCssExtractPlugin = require("mini-css-extract-plugin");

module.exports = {
  ...
```

```
  plugins: [
    new MiniCssExtractPlugin(),
  ],
  module: {
    rules: [
      {
        test: /\.(css|scss)$/,
        use: [MiniCssExtractPlugin.loader, 'css-loader', 'sass-loader']
      }
    ]
  }
}
```

Now, run npm run build and you will see the external files generated in your build folder like main.css.

For automatic switching between using 'style-loader' or MiniCssExtractPlugin depending on the development or production environment. We update the configuration

```
// webpack.config.js
…
module.exports = (env, argv) => {
  const devMode = argv.mode !== 'production';

  return {
    …
    module: {
      rules: [
        {
          test: /\.(css|scss)$/,
          use: [devMode ?  'style-loader' :  MiniCssExtractPlugin.loader, 'css-loader', 'sass-loader']
        }
      ]                          ],
    }
  }
}
```

An example .SCSS file –

```
// src/app.scss
$h1-color: red;
h1 {
  color: $h1-color;
}
```

Next we need to import this into the React component file so that webpack knows that it should process it. Do so by adding the following import to App.jsx

```
// App.jsx
import './app.scss';
const App = () => (
  <h1>Hello World!</h1>
)
```

Now when we run npm run build, webpack will also process the imported SCSS files and output them into a single CSS file (build/main.css).
And this file is injected to index.html page :

```
// run
npm run build
=>
main.js
main.css
index.html

// index.html
<html lang="en">
  <head>
  ...
    <title>Hello World</title>
    <script defer="defer" src="main.js"></script>
    <link href="main.css" rel="stylesheet">
  </head>
...
</html>
```

7)
Adding support for image files ---

We can also set up Webpack to handle image files so that we don't have to manually manage the URLs for the images used in the React components or CSS files.

To test this out, let's create an Icon component that uses an image.

```
// src/Icon.jsx
import reactIcon from './Assets/Images/react.svg';
const Icon = () => (
  <img src={reactIcon} alt="React Logo" />
);
export default Icon;
```

Then extend the existing App component to use this Icon component.

```
// src/App.jsx
import Icon from './Icon.jsx';
const App = () => (
  <div className="container">
    <div className="header">
      <h1>Hello World!</h1>
    </div>
    <div className="react-logo">
      <Icon />
    </div>
  </div>
)
export default App;
```

update the app.scss file

```
// src/app.scss
$h1-color: red;
.container {
  height: 100vh;
  background-color: #E7E3EB;

  .header {
    padding: 15px 20px;
    background-color: #6200EE;
    h1 {
      color: $h1-color;
    }
  }

  .react-logo {
    // background: url('./Assets/Images/react.svg') no-repeat center;
    height: 500px;
    width: 500px;
    // background-size: contain;
    position: absolute;
    top: 50%;
    left: 50%;
    transform: translate(-50%, -50%);
  }
}
```

Extend the Webpack config to handle imported images files
Webpack 5 has built in support for handling imported asset files such as images. So we only need to add a rule telling Webpack to treat image files as asset/resource.

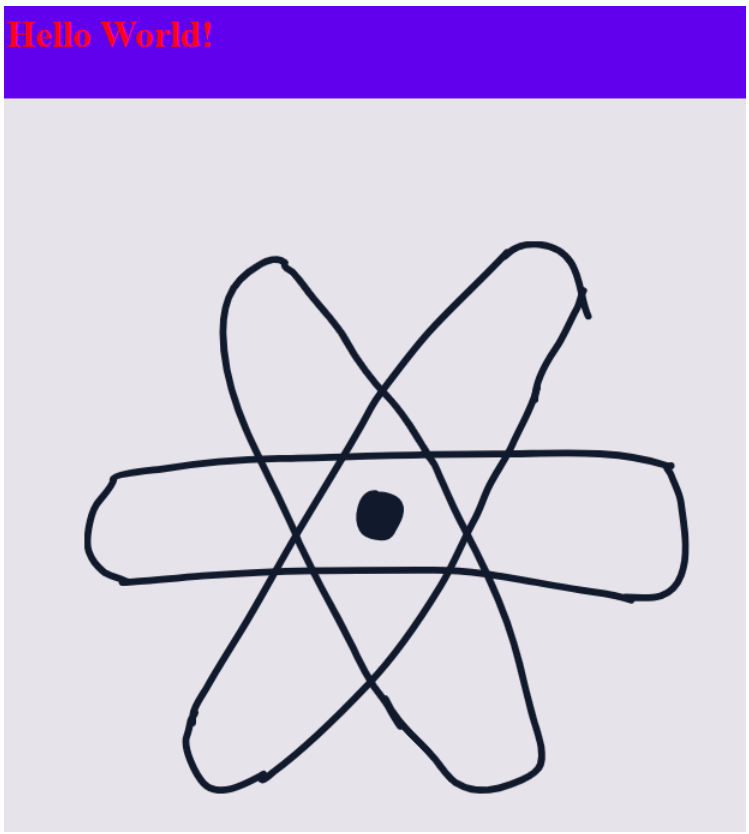
```
module: {
    rules: [
      ...,
      // image
      {
        test: /\.(png|jpe?g|svg)$/i,
        type: 'asset/resource'
      }
    ]
  }
```

Now when we run npm run build, Webpack will also process the imported image files and output them to the output folder.

```
// run
npm run build
=>
59e81412d02b5a205bca.svg
main.js
main.css
index.html
```

It will then adjust the string that is returned from the import statement in the Icon component to point to the path were it outputted the image file.

```
// run web server
cd C:\HONG\JAVASCRIPTtest\HUebung\hongBoilerplate4new
php -S localhost:3000 -t build/
http://localhost:3000/
```



8)
Font Setup --

download Open Sans Italic from google fonts and move them in a folder (src/Assets/Fonts/)
Now create a font.scss file and add the font face inside

```
// src/font.scss
@font-face {
  font-family: 'Open Sans Italic';
  src: url('./Assets/Fonts/OpenSans-Italic.ttf') format('truetype');
  font-weight: normal;
  font-style: italic;
}
```

and let's import the fonts file inside app.scss and apply the fonts to <h1> tag.

```
// src/app.scss
@import url(./font.scss);
…
    h1 {
      font-family: 'Open Sans Italic';
    }
```

Next, update the webpack.config.js to support all the file formats.

```
// webpack.config.js
...
  module: {
    rules: [
      {
        test: /\.(png|jpe?g|svg|ttf)$/
        type: 'asset/resource'
      }
```

Now build

```
// run
npm run build
=>
a9d385277b4833d366ce.ttf
59e81412d02b5a205bca.svg
main.js
main.css
index.html
```

we see new font

9)
Setting up webpack-dev-server ---

The React project we have right now is tedious to use since we need to manually run the build command every time we make changes to code. We also need to manually open the generated files to view them.

We can solve this problem by using webpack-dev-server. It is a Webpack module that sets up a HTTP server that automatically serves the files generated by Webpack so that they can be viewed in a browser. It also watches the source files and automatically rebuilds them when a change is detected. So we can change the code and immediately see the changes in the browser.

Installing and using webpack-dev-server

```
npm install --save-dev webpack-dev-server
```

You can now start the dev server by using npx webpack serve --mode development. We set the mode parameter to development in this case since we want Webpack to generate unminified code that is easier to debug.

```
npx webpack serve --mode development
```

// browser
http://localhost:8080/

We can also add this to package.json as a script so that we can run the more memorable npm start command to start the dev server.

```
// package.json
{
  ...
   "scripts": {
      "start": "webpack serve --mode development",
   }
  ...
}
```

As a last step, we modify the Scripts inside our package.json to include:

```
"start": "webpack-dev-server --mode development --open --hot"
```

This allows us to run npm start to start the Webpack development server. The open and hot options opens our React app in a browser once it's ready and enables hot reloading, respectively.

Or we add property devServer to webpack.config.js

```
// webpack.config.js
module.exports = {
   ...
   devServer: {
      hot: true,   // hot reloading
      port: 3000,  // port on which server will run
      open: true   // open browser automatically on start
   }

}
```

Running npm start will now start the dev server. This will open app in our default browser.

```
npm start
```

CODE ON GIT
https://github.com/hong1234/hongBoilerplate4new

some git commands –

…or create a new repository on the command line –

```
echo "# hongBoilerplate4new" >> README.md
git init
git add README.md
git commit -m "first commit"
git branch -M main
git remote add origin https://github.com/hong1234/hongBoilerplate4new.git
git push -u origin main
```

…or push an existing repository from the command line ---

```
git remote add origin https://github.com/hong1234/hongBoilerplate4new.git
git branch -M main
git push -u origin main
```