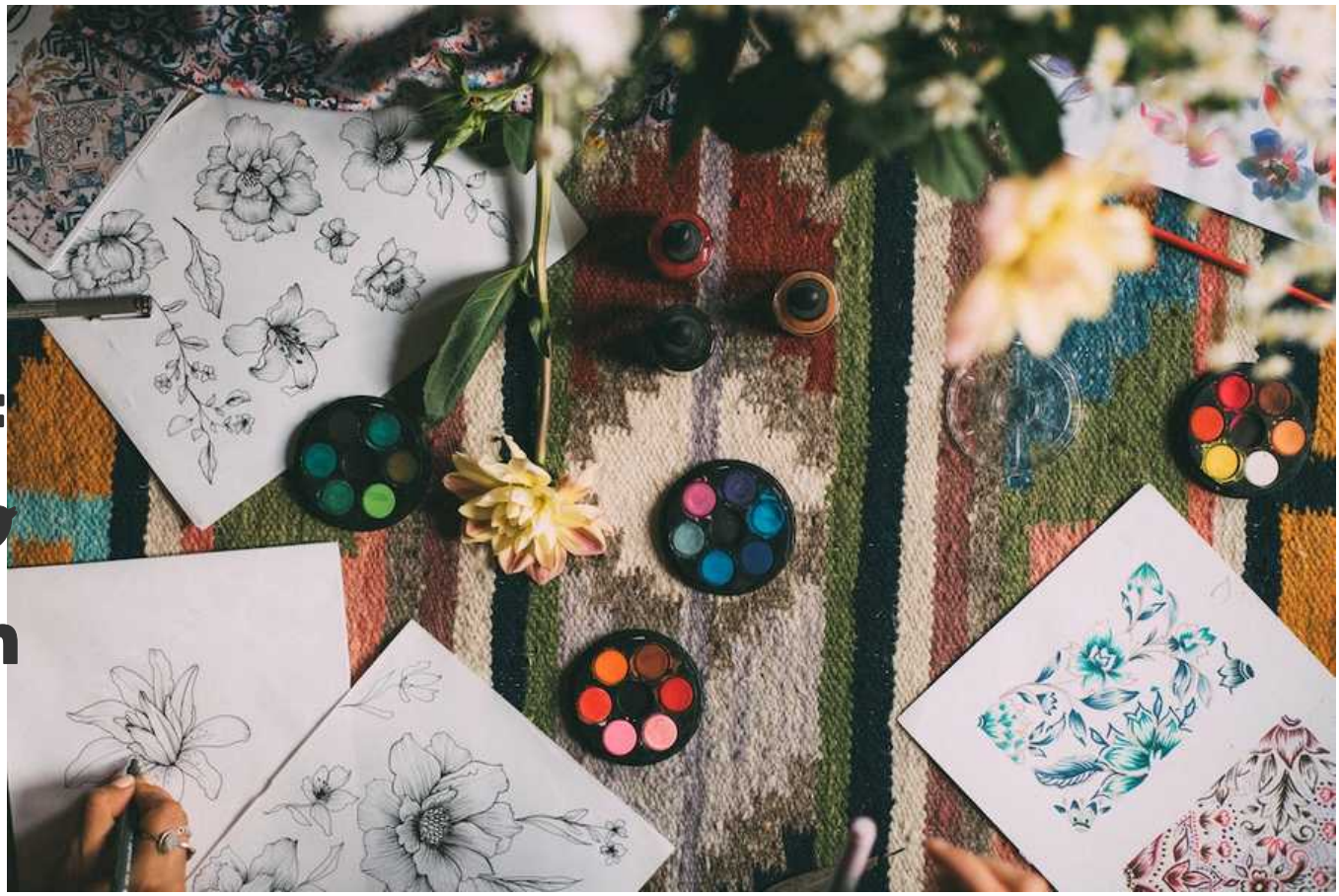
How to Advanced Webpack 5 - Setup Tutorial

OCTOBER 30, 2020 BY ROBIN WIERUCH - EDIT THIS POST



This tutorial is part 3 of 3 in 'Webpack Advanced Setup'-series.

Part 1: How to set up Webpack 5
Part 2: How to set up Webpack 5 with Babel

The previous tutorials have shown you how to set up a basic web application with Webpack 5. So far, Webpack is only used to bundle all your JavaScript files, to transpile new JavaScript features via Babel, and to serve your bundle in development mode via Webpack's Development Server. Basically that's everything that's needed to get started with creating your first web application.

However, Webpack comes with so much more to explore. For instance, eventually you may want to take your project to production. That's when Webpack can help you to build a production ready

bundle which comes with all the optimizations for your source code. In this tutorial, you will learn more about Webpack and how to configure it to your needs. If you don't have a basic Webpack application at your hands, you can take this one from the previous tutorials. The final advanced Webpack setup can be found on GitHub as well.

TABLE OF CONTENTS

WEBPACK'S DEVELOPMENT AND PRODUCTION BUILD

Essentially there are two modes to build your JavaScript application: development and production. You have used the development mode previously to get started with Webpack Dev Server in a local development environment. You can make changes to your source code, Webpack bundles it again, and Webpack Dev Server shows you the recent development build in your browser.

However, eventually you want to have all the build files that are necessary for deploying your web application in production on your web server. Since Webpack bundles all of your JavaScript source code into one *bundle.js* file that is linked in your *dist/index.html* file, you only need essentially these two files on your web server to display your web application for anyone. Let's see how we can create both files for you.

First, you already have the *dist/index.html* file. If you open it, you already see that it uses a *bundle.js* file which is created by Webpack out of all your JavaScript source code files from the *src/* folder.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Hello Webpack bundled JavaScript Project</title>
  </head>
  <body>
```

```
    <div>
      <h1>Hello Webpack bundled JavaScript Project</h1>
    </div>
    <script src="./bundle.js"></script>
  </body>
</html>
```

Second, if you type `npm start`, Webpack will create this *bundle.js* file on the fly which is used for the Webpack Dev Server to start your application in development mode. You never really see the *bundle.js* file yourself.

```
{
  ...
  "scripts": {
    "start": "webpack serve --config ./webpack.config.js --mode development",
    "test": "echo \"Error: no test specified\" && exit 0"
  },
  ...
}
```

Now let's introduce a second npm script to actually build your application for production. We will use Webpack explicitly instead of Webpack Dev Server to bundle all the JavaScript files, reuse the same Webpack configuration from before, but also introduce the production mode:

```
{
  ...
  "scripts": {
    "start": "webpack serve --config ./webpack.config.js --mode development",
    "build": "webpack --config ./webpack.config.js --mode production",
    "test": "echo \"Error: no test specified\" && exit 0"
  },
  ...
}
```

If you run `npm run build`, you will see how Webpack bundles all the files for you. Once the script went through successfully, you can see the *dist/bundle.js* file not generated on the fly, but created for real in your *dist/* folder.

The only thing left for you is to upload your *dist/* folder to a web server now. However, in order to check locally whether the *dist/* folder has everything you need to run your application on a remote web server, use a local web server to try it out yourself:

```
npx http-server dist
```

It should output an URL which you can visit in a browser. If everything works as expected, you can upload the *dist/* folder with its content to your web server. Personally I prefer to use DigitalOcean to host my static websites and web applications.

Also note that Webpack development and production modes come with their own default configuration. Whereas the development mode creates your source code file with an improved developer experience in mind, the production build does all the optimizations to your source code.

Exercises:

- Get comfortable with http-server to try your production ready web application locally
- Host your web application somewhere (e.g. DigitalOcean)

HOW TO MANAGE YOUR WEBPACK BUILD FOLDER

Every time you run `npm run build`, you will see Webpack creating a new version of your bundle JavaScript source code with a *dist/bundle.js* file. Eventually your Webpack build pipeline will become more complex and you end up with more than two files in your *dist/* folder. Suddenly the folder becomes a mess, because you don't know which files belong to the most recent build. The best thing would be to start with an empty *dist/* folder with every Webpack build.

Let's say we wanted to wipe our *dist/* folder with every Webpack build. It would mean that our auto generated *dist/bundle.js* file would be removed (good), but also our *dist/index.html* file which we implemented manually (bad). We don't want to re-create this file by hand for every Webpack build again. In order to auto generate the *dist/index.html* file as well, we can use a Webpack plugin. First, install the html-webpack-plugin plugin as dev dependency from your project's root directory:

```
npm install --save-dev html-webpack-plugin
```

After a successful installation, introduce the Webpack plugin in your Webpack *webpack.config.js* file:

```
const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');

module.exports = {
  entry: path.resolve(__dirname, './src/index.js'),
  module: {
    rules: [
      {
        test: /\.(js)$/,
        exclude: /node_modules/,
        use: ['babel-loader'],
```

```
      },
    ],
  },
  resolve: {
    extensions: ['*', '.js'],
  },
  output: {
    path: path.resolve(__dirname, './dist'),
    filename: 'bundle.js',
  },
  plugins: [new HtmlWebpackPlugin()],
  devServer: {
    contentBase: path.resolve(__dirname, './dist'),
  },
};
```

Now, run `npm run build` again and see how it auto generates a new *dist/index.html* file. It comes with a default template for how the file should be structured and what should be in the file. However, if you want to have custom content for your *dist/index.html* file, you can specify a template yourself:

```
const HtmlWebpackPlugin = require('html-webpack-plugin')

module.exports = {
  ...
  plugins: [
    new HtmlWebpackPlugin({
      title: 'Hello Webpack bundled JavaScript Project',
      template: path.resolve(__dirname, './src/index.html'),
    })
  ],
  ...
};
```

Then, create a new *src/index.html* template file in your source code folder and give it the following content:

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= htmlWebpackPlugin.options.title %></title>
  </head>
  <body>
    <div>
      <h1><%= htmlWebpackPlugin.options.title %></h1>

      <div id="app">
    </div>
  </body>
</html>
```

Note that you don't need to specify the script tag with the *bundle.js* file anymore, because Webpack will introduce it automatically for you. Also note that you don't need necessarily the `id` attribute and the div container, but we have used in the previous tutorial to execute some JavaScript on it.

Now, run `npm run build` again and see whether the new auto generated *dist/index.html* matches your template from *src/index.html*. Finally we have been able to create both files, *dist/bundle.js* and *dist/index.html* automatically with Webpack. This means we can delete the content of our *dist/* folder with every Webpack build. In order to do so, introduce the clean-webpack-plugin plugin:

```
npm install --save-dev clean-webpack-plugin
```

Then introduce it in your *webpack.config.js* file:

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

module.exports = {
  ...
  plugins: [
    new CleanWebpackPlugin(),
    new HtmlWebpackPlugin({
      title: 'Hello Webpack bundled JavaScript Project',
      template: path.resolve(__dirname, './src/index.html'),
    }),
  ],
  ...
};
```

Now, every Webpack build will wipe the content of your *dist/* folder before creating the new *dist/index.html* and *dist/bundle.js* files from scratch. Having it set up this way, you will never find files from older Webpack builds in your *dist/* folder which is perfect for just taking your entire *dist/* folder to production.

Note: If you are using a version control system like GitHub, you can put the build folder (dist/) into your .gitignore file, because everything is auto generated for everyone anyway. After someone got a copy of your project, the person can do a `npm run build` to generate the files.

Exercises:

- Watch your *dist/* folder when running `npm run build`
- Adjust the *src/index.html* to your needs
- Check out more optional configuration that are available for your new Webpack plugins

WEBPACK SOURCE MAPS

Webpack bundles all of your JavaScript source code files. That's perfect, however, it introduces one pitfall for us as developers. Once you introduce a bug and see it in your browser's developer tools, it's often difficult to track down the file where the bug happened, because everything is bundled into one JavaScript file by Webpack. For instance, let's say our *src/index.js* file imports a function from another file and uses it:

```
import sum from './sum.js';

console.log(sum(2, 5));
```

In our *src/sum.js*, we export this JavaScript function but unfortunately introduced a typo in it:

```
export default function (a, b) {
  return a + c;
};
```

If you run `npm start` and open the application in your browser, you should see the error happening in your developer tools:

```
sum.js:3 Uncaught ReferenceError: c is not defined
    at eval (sum.js:3)
    at eval (index.js:4)
    at Module../src/index.js (bundle.js:457)
    at __webpack_require__ (bundle.js:20)
    at eval (webpack:///multi_(:8080/webpack)-dev-server/client?:2:18)
    at Object.0 (bundle.js:480)
    at __webpack_require__ (bundle.js:20)
    at bundle.js:84
    at bundle.js:87
```

If you click on the *sum.js* file where the error happened, you only see Webpack's bundled output. In the case of this example, it's still readable, however imagine the output for a more complex problem:

```
__webpack_require__.r(__webpack_exports__);
/* harmony default export */ __webpack_exports__["default"] = (function (a, b)
  return a + c;
});
;
```

Take this one step further and introduce the bug in your Webpack build for production instead. Run `npm run build` and `npx http-server dist` to see the error in your browser again:

```
bundle.js:1 Uncaught ReferenceError: c is not defined
    at Module.<anonymous> (bundle.js:1)
    at t (bundle.js:1)
    at bundle.js:1
    at bundle.js:1
```

This time it's hidden in your *bundle.js* file without letting you know about the actual file that's causing it. In addition, once you click on the *bundle.js* file, you only see Webpack's bundled JavaScript for production which is not in a readable format.

In conclusion, it's not a great developer experience, because it becomes more difficult with Webpack's bundled JavaScript files to track down errors. That's true for development mode, but even more for production mode.

In order to overcome this problem, there are source maps which can be introduced to give Webpack a reference to the origin source code. By using the source maps, Webpack can map all the bundled source code back to the original source. In your *webpack.config.js* file, introduce one common configuration for source maps:

```
...

module.exports = {
  ...
  devtool: 'source-map',
};
```

Afterward, with the bug still in your source code, run `npm run build` and `npx http-server dist` again. In your browser, note how the bug can be tracked down to the causing file *sum.js*:

```
sum.js:2 Uncaught ReferenceError: c is not defined
    at Module.<anonymous> (sum.js:2)
    at t (bootstrap:19)
    at bootstrap:83
    at bootstrap:83
```

Clicking on the file gives you the actual source code and location of the bug even though all your JavaScript source code got bundled by Webpack. Also note that there is a new file called *dist/bundle.js.map* which is used to perform the mapping between actual source code from *src/* and the bundled JavaScript in *dist/bundle.js*.

Exercises:

- Introduce a few bugs in your source code and bundle your project without and with source maps to see the difference in your browser's developer tools

---

WEBPACK DEVELOPMENT/BUILD CONFIGURATION

So far, we have used one common Webpack configuration for development and production. However, we can introduce a configuration for each mode as well. In your *package.json*, change the start and build scripts to the following:

```
{
  ...
  "scripts": {
    "start": "webpack serve --config ./webpack.dev.js",
    "build": "webpack --config ./webpack.prod.js",
    "test": "echo \"Error: no test specified\" && exit 0"
  },
  ...
}
```

Now create these two new files, copy and paste the old *webpack.config.js* configuration over to both of them, and delete the old *webpack.config.js* file afterward. Next, since we have omitted the Webpack modes in the npm scripts, introduce them again for each of your Webpack configuration files. First, the *webpack.dev.js* file:

```
...

module.exports = {
  mode: 'development',
  ...
};
```

Second, the *webpack.prod.js* file:

```
...

module.exports = {
  mode: 'production',
  ...
};
```

Your npm scripts to start and build your application should work again. But you may wonder: What's the difference now? Except for the Webpack modes which we passed in dynamically before, the Webpack configuration is the same for development and production. We have even introduced unnecessary duplication. More about the latter one later.

In a growing Webpack configuration, you will introduce things (e.g. plugins, rules, source maps) which should behave differently for development and production. For instance, let's take the source maps which we have implemented previously. It's a performance heavy process to create source map files for a large code base. In order to keep the development build operating fast and efficient for a great developer experience, you want to have your source maps in development not 100% effective as the source maps from your production build. It should be faster to create them for development mode. That's why you can introduce your first change for the *webpack.dev.js* file which is not reflected in your production configuration:

```
...

module.exports = {
  mode: 'development',
  ...
  devtool: 'eval-source-map',
};
```

Now, your source maps are generated differently for your development and production modes, because they are defined in different ways in your two Webpack configuration files. This was only one instance of having a different configuration for Webpack in development and production. In the future, you will introduce more of them and be happy to have to separate places for them.

Exercises:

- Visit Webpack's documentation to find out more about the different source map options

---

WEBPACK MERGE CONFIGURATION

At the moment, your Webpack configuration files for development and production share lots of common configuration. What if we would be able to extract the common configuration to a separate yet commonly used file and only choose extra specific configuration based on the development and production? Let's do it by adjusting our *package.json* file:

```
{
  ...
  "scripts": {
```

```
    "start": "webpack serve --config build-utils/webpack.config.js --env env=(
    "build": "webpack --config build-utils/webpack.config.js --env env=prod",
    "test": "echo \"Error: no test specified\" && exit 0"
  },
  ...
}
```

As you can see, we reference a new shared *webpack.config.js* for both npm scripts. The file is located in a new *build-utils* folder. In order to distinguish the running scripts later in the Webpack configuration, we pass an environment flag (dev, prod) to the configuration as well.

Now, create the shared *build-utils/webpack.config.js* file again, but this time in the new dedicated *build-utils* folder, and give it the following configuration:

```
const { merge } = require('webpack-merge');

const commonConfig = require('./webpack.common.js');

module.exports = ({ env }) => {
  const envConfig = require(`./webpack.${env}.js`);

  return merge(commonConfig, envConfig);
};
```

You can see that the function receives our env environment flag from the npm script. That way, we can dynamically require a environment specific Webpack configuration file with JavaScript template literals and merge it with a common Webpack configuration. In order to merge it, let's install a little helper library:

```
npm install --save-dev webpack-merge
```

Next, we have to implement three files in the *build-utils* folder now:

- webpack.common.js: shared Webpack configuration for development and build mode.
- webpack.dev.js: Webpack configuration only used by development mode.
- webpack.prod.js: Webpack configuration only used by production mode.

Let's start with the shared Webpack configuration in a new *build-utils/webpack.common.js* file:

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

module.exports = {
```

```
  entry: path.resolve(__dirname, '..', './src/index.js'),
  module: {
    rules: [
      {
        test: /\.(js)$/,
        exclude: /node_modules/,
        use: ['babel-loader']
      }
    ]
  },
  resolve: {
    extensions: ['*', '.js']
  },
  plugins: [
    new CleanWebpackPlugin(),
    new HtmlWebpackPlugin({
      title: 'Hello Webpack bundled JavaScript Project',
      template: path.resolve(__dirname, '..', './src/index.html'),
    })
  ],
  output: {
    path: path.resolve(__dirname, '..', './dist'),
    filename: 'bundle.js'
  },
  devServer: {
    contentBase: path.resolve(__dirname, '..', './dist'),
  },
};
```

Note that some file paths have changed in contrast to the previous Webpack configuration, because we have this file in a dedicated folder now. Also note that there are no Webpack modes and no source maps anymore. These two options will become environment (e.g. development, production) specific in their dedicated Webpack configuration files.

Move on by creating the *build-utils/webpack.dev.js* file and give it the following content:

```
module.exports = {
  mode: 'development',
  devtool: 'eval-source-map',
};
```

Last but not least, the new *build-utils/webpack.prod.js* file which receives the following content:

```
module.exports = {
  mode: 'production',
  devtool: 'source-map',
};
```

Your folder structure should be similar to the following now. Note that there are no Webpack configurations outside of the *build-utils/* folder from previous sections anymore:

```
- build-utils/
-- webpack.common.js
-- webpack.config.js
-- webpack.dev.js
-- webpack.prod.js
- dist/
-- bundle.js
-- bundle.js.map
-- index.html
- src/
-- index.html
-- index.js
- package.json
- .babelrc
```

That's it. Your `npm start` and `npm run build` scripts should work now. Both are working with different configuration for Webpack mode and source maps in respect to their *build-utils/webpack.dev.js* and *build-utils/webpack.prod.js* configuration files. But they also share a common Webpack configuration from *build-utils/webpack.common.js*. Everything is dynamically merged in your *build-utils/webpack.config.js* file which does the dynamic merging based on the incoming flags from your npm scripts in the *package.json*.

Exercises:

- Revisit your *build-utils/* folder with all its files and the *package.json* file
  - Understand how the commands flow from *package.json* to all the files in the *build-utils/* folder
  - Understand how your Webpack configuration gets merged in the *build-utils/webpack.config.js*

WEBPACK ENVIRONMENT VARIABLES: DEFINITION

Sometimes you may want to know in your source code whether you are in development or production mode. For these cases you can specify dynamic environment variables via Webpack. Since you have a Webpack configuration file for each environment (dev, prod), you can define dedicated environment variables for them. In your *build-utils/webpack.dev.js*, define a environment variable the following way:

```
const { DefinePlugin } = require('webpack');

module.exports = {
  mode: 'development',
```

```
  plugins: [
    new DefinePlugin({
      'process.env': {
        'NODE_ENV': JSON.stringify('development'),
      }
    }),
  ],
  devtool: 'eval-source-map',
};
```

The same applies to your *build-utils/webpack.prod.js* file, but with a different environment variable:

```
const { DefinePlugin } = require('webpack');

module.exports = {
  mode: 'production',
  plugins: [
    new DefinePlugin({
      'process.env': {
        'NODE_ENV': JSON.stringify('production'),
      }
    }),
  ],
  devtool: 'source-map',
};
```

Now you can use (e.g. `console.log(process.env.NODE_ENV);`) the environment variable in your *src/index.js* file or any other JavaScript in your *src/* folder to make decisions based on it. In this case, you have created two different environment variables -- each in respect to the Webpack mode. However, in the future you may introduce more environment variables for certain scenarios.

Exercises:

- Think about other scenarios where environment variables can be used
- Is it secure to use sensitive information in environment variables when they are exposed in your Webpack configuration files?

WEBPACK ENVIRONMENT VARIABLES: .ENV

Previously you started to define your environment variables in your Webpack configuration files. However, that's not the best practice for sensitive information. For instance, let's say you want to use API keys/secrets (credentials) to access your database based on your development or production environment. You wouldn't want to expose these sensitive information in your Webpack configuration which may be shared with others. Instead, you would want to introduce dedicated

files for your environment files which can be kept away from others and version control systems like Git or SVN.

Let's start by creating two environment variables files for development and production mode. The first one is for development mode and is called *.env.development*. Put it in your project's root directory with the following content:

```
NODE_ENV=development
```

The second one is called *.env.production* and has some other content. It's also placed in your project's root directory:

```
NODE_ENV=production
```

By using the dotenv-webpack plugin, you can copy these environment variables into your Webpack configuration files. First, install the plugin:

```
npm install dotenv-webpack --save-dev
```

Second, use it in your *build-utils/webpack.dev.js* file for the development mode:

```
const path = require('path');
const Dotenv = require('dotenv-webpack');

module.exports = {
  mode: 'development',
  plugins: [
    new Dotenv({
      path: path.resolve(__dirname, '..', './.env.development'),
    })
  ],
  devtool: 'eval-source-map',
};
```

And third, use it in your *build-utils/webpack.prod.js* file for the production mode:

```
const path = require('path');
const Dotenv = require('dotenv-webpack');

module.exports = {
  mode: 'development',
  plugins: [
    new Dotenv({
```

```
      path: path.resolve(__dirname, '..', './.env.production'),
    })
  ],
  devtool: 'eval-source-map',
};
```

Now you can introduce sensitive information -- such as IP addresses, account credentials, and API keys/secrets -- in your environment variables via your *.env.development* and *.env.production* files. Your Webpack configuration will copy them over to make them accessible in your source code (see previous section). Don't forget to add these new *.env* files to your *.gitignore* -- if you are using version control systems (e.g. Git) -- to hide your sensitive information from third parties.

Exercises:

- Create a *.gitignore* file to ignore your environment variable files in case you are planning to use Git

---

WEBPACK ADDONS

Webpack has a large ecosystem of plugins. Several of them are already used implicitly by using Webpack development or production modes. However, there are also other Webpack plugins which improve your Webpack bundle experience. For instance, let's introduce addons which can be used optionally to analyze and visualize your Webpack bundle. In your *package.json*, introduce a new npm script for your build process, but this time with Webpack addons:

```
{
  ...
  "scripts": {
    "start": "webpack serve --config build-utils/webpack.config.js --env env=
    "build": "webpack --config build-utils/webpack.config.js --env env=prod",
    "build:analyze": "npm run build -- --env addon=bundleanalyze",
    "test": "echo \"Error: no test specified\" && exit 0"
  },
  ...
}
```

Note how this new npm script runs another npm script but with additional configuration (here Webpack addons). However, the Webpack addons will not run magically. In this case, they are only passed as flags to our Webpack configuration. Let's see how we can use them in our *build-utils/webpack.config.js* file:

```
const { merge } = require('webpack-merge');
```

```
const commonConfig = require('./webpack.common.js');

const getAddons = (addonsArgs) => {
  const addons = Array.isArray(addonsArgs)
    ? addonsArgs
    : [addonsArgs];

  return addons
    .filter(Boolean)
    .map((name) => require(`./addons/webpack.${name}.js`));
};

module.exports = ({ env, addon }) => {
  const envConfig = require(`./webpack.${env}.js`);

  return merge(commonConfig, envConfig, ...getAddons(addon));
};
```

Now, not only the common and environment specific Webpack configuration get merged, but also the optional addons which we will put into a dedicated *build-utils/addons* folder. Let's start with the *build-utils/addons/webpack.bundleanalyze.js* file:



```
const path = require('path');
const { BundleAnalyzerPlugin } = require('webpack-bundle-analyzer');

module.exports = {
  plugins: [
    new BundleAnalyzerPlugin({
      analyzerMode: 'static',
      reportFilename: path.resolve(
        __dirname,
        '..',
        '..',
        './dist/report.html'
      ),
      openAnalyzer: false,
    }),
  ],
};
```

Next, install the Webpack addon via npm on the command line:

```
npm install --save-dev webpack-bundle-analyzer
```

As you can see, you have introduced a specific Webpack addon, which can be optionally added, in a new *build-utils/addons/* folder. The naming of the addon files matches the passed flag from the

npm script in your *package.json*. Your Webpack merge makes sure to add all passed addon flags as actual addons to your Webpack configuration.

Now try the optional tool for Webpack analytics and visualization yourself. On your command line, type `npm run build:analyze`. Afterward, check your *dist/* folder for new files. You should find one which you can open the following way:

- Webpack's bundleanalyze: *dist/report.html*
  - open via `npx http-server dist`, vist the URL, and append */report.html*

You will see your build optimized Webpack bundle with two different visualizations. You don't have much code in your application yet, but once you introduce more source code and more external libraries (dependencies) with your node package manager, you will see how your Webpack bundle will grow in size. Eventually you will introduce a large library by accident which makes your application too big. Then both analytic and visualization tools can help you to find this culprit.

Exercises:

- Install a library like lodash to your application, import it, and use a function from it in your source code
  - Run again `npm run build:analyze` and check both visualizations
  - You should see that Lodash makes up a huge part of your Webpack bundle whereas your actual source code takes up only a minimal part
- Explore more Webpack addons and introduce them to your application
  - You can also come up with more npm scripts for different Webpack addons
- Optional React: Check out the minimal React with Webpack setup
  - Put it into your advanced Webpack application
  - Confirm your final result with the official advanced React with Webpack setup

. . .

You have seen how Webpack can be used to bundle your web application in a sophisticated way. It helps you to automate certain steps and to optimize your build process. You can find the finished project on GitHub. If you have any other internal plugins or libraries that you are using with Webpack, let me know about them in the comments below.

Continue Reading: How to set up React with Webpack and Babel

Show Comments

KEEP READING ABOUT WEBPACK›

HOW TO WEBPACK 5 - SETUP TUTORIAL

Webpack is a JavaScript bundler for your web application. In the past, you had to link JavaScript files manually in HTML files. Nowadays, Webpack takes care about it. In this tutorial, I want to walk...

HOW TO REACT ROUTER WITH WEBPACK 5 - SETUP TUTORIAL

If you happen to have a custom Webpack setup, you may be wondering how to set up React Router with Webpack. Let's say we have the following minimal React application using React Router: If you open...

TAKE PART

NEVER MISS AN ARTICLE ABOUT WEB DEVELOPMENT AND JAVASCRIPT.

- ✔ Join 50.000+ Developers
- ✔ Learn Web Development with JavaScript
- ✔ Tips and Tricks
- ✔ Access Tutorials, eBooks and Courses
- ✔ Personal Development as a Software Engineer

Your email address SUBSCRIBE

View our Privacy Policy.

PORTFOLIO

Online Courses

Open Source

Tutorials

ABOUT

About me

What I use

How to work with me

How to support me

Contact Me Privacy & Terms