4/25/2022

Nagendra Mirajkar

AES Zensar

Webpack

AES Zensar

Table of Contents

[What is webpack? 1](#_Toc101785807)

[Installing webpack 2](#_Toc101785808)

[Setting up a project to bundle 4](#_Toc101785809)

[Installing scripts to bundle 5](#_Toc101785810)

[Creating a bundle 6](#_Toc101785811)

[Configuring webpack to generate HTML 7](#_Toc101785812)

[Enabling development mode in webpack 11](#_Toc101785813)

[Cleaning up the dist folder 12](#_Toc101785814)

[Running webpack with an npm script 14](#_Toc101785815)

[Installing and running a server with hot reload 15](#_Toc101785816)

[Final example of a webpack implementation 17](#_Toc101785817)

[Wrapping up this webpack tutorial 19](#_Toc101785818)

**What is webpack?**



[webpack](https://webpack.js.org/) is an open-source JavaScript module bundler. At [more than 50k stars](https://github.com/webpack/webpack) on GitHub, it’s the leader in this space.

Webpack allows you to split your JavaScript into separate modules in development (better for maintenance) while letting you compile those modules into a single bundle in production (better for performance).

Historically, many new or less experienced developers have shied away from webpack due to its complexity. That still might be true to an extent, but webpack continues to receive regular updates and is getting better with every version. The webpack team released [Version 5.0.0](https://webpack.js.org/blog/2020-10-10-webpack-5-release/) in October 2020 and there have been numerous incremental updates since then.

Notably, [webpack 4.0.0](https://v4.webpack.js.org/) was the first version of webpack that doesn’t require a webpack.config.js file to bundle your project. This alone has made it easier for newer developers to start using it.

This webpack tutorial will focus mainly on the latest major version (5.x.x as of this writing). You can look more into the history of webpack through their [changelog](https://github.com/webpack/webpack/releases) but let’s get to how you can use webpack in a project today.

## Installing webpack

To install webpack 5 requires Node 10.13.0 or higher, so if you haven’t updated Node in a while you’ll have to do that before you install webpack.

The webpack docs strongly recommend installing webpack locally, rather than globally. This means you would install it separately on each project rather than having a single global installation that’s used on every project. With individual local installs you’ll be able to upgrade (or not upgrade) each install as needed.

To follow along with this webpack tutorial, create a project directory that you’ll use to run the various commands I’ll be covering. Once the project directory is ready, you’ll run the following two commands (the first one ensures you’re inside the directory):

cd webpack-example

npm init -y

In this case, I’ve named my example project’s root folder webpack-example. You can name yours whatever you want. The npm init -y command initializes the directory with npm’s default settings, which creates a package.json file.

Next, I’ll install webpack and the webpack CLI

npm install webpack webpack-cli --save-dev

Once I’ve installed both packages as dependencies for the project, my package.json file will look something like this:

{

"name": "webpack-example",

"version": "1.0.0",

"description": "",

"main": "index.js",

"scripts": {

"test": "echo \"Error: no test specified\" && exit 1"

},

"keywords": [],

"author": "",

"license": "ISC",

"devDependencies": {

"webpack": "^5.27.2",

"webpack-cli": "^4.5.0"

}

}

Notice the two packages (webpack and webpack-cli) now listed as dependencies near the bottom. This project is now more or less ready to start using webpack as its bundler, so let’s put it into action in this webpack tutorial.

## Setting up a project to bundle

In its current state, this project has nothing to bundle yet. So let’s add some stuff to it to demonstrate how a simple bundle process happens. This alone should be enough to get you started with webpack, but afterward I’ll go into some configurations you can add to make webpack even more powerful.

In the same location as my package.json file, I’m going to add the following:

* A folder called src
* An index.html file inside src
* An index.js file inside src
* A dist folder

Go ahead and add those if you want to follow along with the steps in this webpack tutorial. The webpack docs explain nicely what the src (source) and dist (distribution) folders are for (though this is not specific to webpack but more about bundling and build processes in general):

*The “source” code is the code that we’ll write and edit. The “distribution” code is the minimized and optimized output of our build process that the browser will display.*

Ideally, my bundle setup will empty the dist folder each time I create a bundle. I’ll cover more on this a little later but for now I’m going to focus on editing files in the src folder (which is where the editing always takes place).

To begin, I need to add some content to both index.html and index.js. The index.html file, naturally, can contain any content. I’ll customarily include all [website](https://www.codeinwp.com/blog/how-to-make-a-wordpress-website/) content in the src directory, including stylesheets, [images](https://www.codeinwp.com/blog/how-to-optimize-images/), etc., but this simple example is for demonstrating webpack’s features.

Traditionally, when you want to add one or more libraries as dependencies in a project, you would list them at the bottom of your index.html page using separate <script> elements, one after another. You’ll also include your custom JavaScript that uses those other dependencies. That’s where a tool like webpack helps because you’ll not only avoid having to manually add scripts to your pages but you’ll be able to add them, bundle them for optimization, and sometimes even load them on demand.

## Installing scripts to bundle

Instead of the traditional and nonoptimal approach to adding and incorporating scripts, I’m going to use npm to install my dependencies, then webpack to bundle them. For demonstration purposes, I’m going to use two JavaScript utility libraries:

* [Flicking](https://github.com/naver/egjs-flicking) – A JavaScript carousel
* [Panzoom](https://github.com/anvaka/panzoom) – A pan/zoom framework

To be clear: webpack doesn’t require these; they are a few example utilities that I selected pretty much at random to demonstrate webpack’s bundling features. Your project will include different libraries and utilities, likely larger tools like [React, Vue](https://www.codeinwp.com/blog/angular-vs-vue-vs-react/), or Babel.js for [cross-browser](https://www.codeinwp.com/blog/best-cross-browser-testing-tools/) JavaScript.

To use the utilities I chose, I have to install them, so I’m going to do that first:

npm install panzoom --save

npm install @egjs/flicking --save

Code language: CSS (css)

In this case, I’m using the --save flag instead of --save-dev because I want these as part of my production build. When I installed webpack, I installed it as a developer dependency, so webpack won’t be part of my production build.

Now my package.json has the following appended below the devDependencies section:

"dependencies": {

"@egjs/flicking": "^3.8.1",

"panzoom": "^9.4.1"

}

Code language: JavaScript (javascript)

I can then include in my index.html file elements that my two utilities will interact with. For example, the carousel would require a wrapper and “panel” elements in the HTML while the zoom utility requires one or more HTML elements that the utility’s API would target.

To demonstrate that these utilities are successfully bundled, I’m going to add the following to my index.js file in the src folder:

import panzoom from 'panzoom';

import flicking from '@egjs/flicking';

console.log(panzoom);

console.log(flicking);

Code language: JavaScript (javascript)

Behind the scenes, webpack will recognize the two import statements and will look for those two dependencies in my node\_modules folder. The console() commands are merely to demonstrate that the bundle was successful and both imports are working as expected.

## Creating a bundle

Webpack uses a specific JavaScript file, named in package.json, as something called an entry point. The entry point indicates to webpack which module to use to build out the project’s dependency graph. A dependency graph is essentially a map of every module your application needs.

I can specify a custom entry point if I want, but I prefer to use webpack with as little configuration as needed, so I’m going to go with the default entry point: ./src/index.js. For info on changing the entry point, [see the docs](https://webpack.js.org/concepts/#entry).

Now that I’ve installed a few dependencies, I can build my bundle with webpack. I can do this by running the following command in my project’s directory:

npx webpack

Running this command tells webpack to bundle my JavaScript dependencies using the specified entry point and the dist folder will produce the output. In this case, main.js is the only generated file that my app will use. Even though I included an index.html file in the src folder, webpack hasn’t done anything with that yet, so webpack has only produced main.js so far.

If I open up main.js in the dist folder, I’ll find a minified file containing all the dependencies specified in my project (in this case, the two utilities and any utilities those two depend on).

⚠️ Note: If you get a warning in your CLI about the *mode* option not being set, don’t worry about that for now. I’ll show you how to correct that later.

## Configuring webpack to generate HTML

As mentioned, my index.html file doesn’t appear in the dist directory during the bundle process. I can do that manually if I want, but that defeats the purpose of using a tool like webpack to streamline my builds.

To help with this, I’m going to install a plugin called [HtmlWebpackPlugin](https://webpack.js.org/plugins/html-webpack-plugin/" \t "_blank):

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

My devDependencies in package.json will reflect the change:

"devDependencies": {

"html-webpack-plugin": "^5.3.1",

"webpack": "^5.27.2",

"webpack-cli": "^4.5.0"

},

Code language: JavaScript (javascript)

Although I can use webpack without any manual configuration, it works best with some configuration. To add that manual configuration, I’m going to create a webpack.config.js file in my project’s root folder. Inside this file, I’m going to add the following to utilize this newly installed plugin:

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

const path = require('path');

module.exports = {

plugins: [

new HtmlWebpackPlugin()

]

};

Code language: JavaScript (javascript)

With that in place, I’ll run npx webpack again for a better result in my dist folder. The HtmlWebpackPlugin generates a minified HTML file that adds my JavaScript bundle, referenced in a <script> tag in the page.

The problem is, this isn’t using the index.html file that I originally created in my src folder. Let’s configure HtmlWebpackPlugin to use that file as a template instead of building its own file.

I’m going to make the following changes to src/index.html:

<!DOCTYPE html>

<html lang="en">

<head>

<title><%= htmlWebpackPlugin.options.title %></title>

<meta charset="UTF-8" />

<meta name="viewport" content="width=device-width,initial-scale=1" />

<meta name="description" content="<%= htmlWebpackPlugin.options.metaDesc %>" />

</head>

<body>

<h1><%= htmlWebpackPlugin.options.header %></h1>

<div id="wrapper" style="height: 120px">

<div class="panel"></div>

<div class="panel"></div>

<div class="panel"></div>

</div>

<div id="zoom-scene"></div>

</body>

</html>

Code language: HTML, XML (xml)

Notice I’ve added multiple variables that are accessing values, used here as placeholders. This is similar to what’s done with JavaScript templating or a backend templating language. Notice there’s no reference to a JavaScript file.

Next I’m going to add some options to HtmlWebpackPlugin() in my webpack.config.js file. The plugins: [] section of my webpack.config.js file now looks like this:

plugins: [

new HtmlWebpackPlugin({

hash: true,

title: 'Webpack Example App',

header: 'Webpack Example Title',

metaDesc: 'Webpack Example Description',

template: './src/index.html',

filename: 'index.html',

inject: 'body'

})

]

Code language: JavaScript (javascript)

Note in the code above:

* I have hash set to true for cache busting
* I’ve specified the location of my template
* I’ve given the name of the HTML file to output inside the dist folder

Additionally, notice I’ve specified the three variables used as placeholders in my template. I can create whatever properties I want here, as long as they don’t conflict with any of the [predefined options](https://github.com/jantimon/html-webpack-plugin#options) the plugin allows. Finally, I’ve included the inject option set to a value of body to ensure the script appears at the bottom of the <body> element in my HTML.

## Enabling development mode in webpack

Currently, when webpack builds my dist folder, it’s building and bundling it using production mode, which is the default. If I’m continuously building and checking my work locally, it might be more useful if my files aren’t minified. Additionally, my specifying a mode, I’ll avoid the warning I mentioned earlier.

To enable development mode I’ll add a line to my webpack.config.js file so the complete file will now look like this:

const path = require('path');

module.exports = {

plugins: [

new HtmlWebpackPlugin({

hash: true,

title: 'Webpack Example App',

header: 'Webpack Example Title',

metaDesc: 'Webpack Example Description',

template: './src/index.html',

filename: 'index.html',

inject: 'body'

})

],

mode: 'development'

};

Code language: JavaScript (javascript)

Notice the mode that’s now specified in my module.exports object (don’t forget the comma after the square bracket). When I’m ready to publish my work, I can switch this to mode: production. Now my builds will take place cleanly without that warning message I mentioned earlier.

## Cleaning up the dist folder

With this setup, whenever I run webpack on my project, the dist folder remains the same with webpack creating some files and overwriting others. That’s not bad, but ideally I’d like each build to output to an empty dist folder.

This makes sense because some builds might have new directories, new files to build, renamed files, and so forth. I don’t want the old files hanging around, I only want whatever webpack builds and bundles each time it runs.

I’m going to add another line to my webpack.config.js file, which will now look like this (again, note the additional comma for proper syntax):

const path = require('path');

module.exports = {

plugins: [

new HtmlWebpackPlugin({

hash: true,

title: 'Webpack Example App',

header: 'Webpack Example Title',

metaDesc: 'Webpack Example Description',

template: './src/index.html',

filename: 'index.html',

inject: 'body'

})

],

mode: 'development',

output: {

clean: true

}

};

Code language: JavaScript (javascript)

Now module.export includes an output object that has a single property/value pair: clean: true. This ensures webpack will clean my dist folder before each build.

## Running webpack with an npm script

Up to this point, I’ve been using the npx webpack command to bundle my resources. That’s one way to do it, but working with webpack is more efficient when executing the commands using npm scripts. This will come in handy when I want to add other commands as part of my build process.

In my package.json file, there’s a “scripts” section. I’m going to add a line to that so it looks like this (again, note the extra comma):

"scripts": {

"test": "echo \"Error: no test specified\" && exit 1",

"build": "webpack"

},

Code language: JavaScript (javascript)

Now I can run webpack using the following command in my project’s root directory:

npm run build

This has the same result as the npx webpack command: Webpack will use the entry point inside src to begin my build, it will bundle my dependencies, and it will generate the output in the dist folder.

Alternatively, if I want to use scripts to differentiate between development and production builds, I can do the following:

"scripts": {

"test": "echo \"Error: no test specified\" && exit 1",

"dev": "webpack --mode development",

"build": "webpack --mode production"

},

Code language: JavaScript (javascript)

Now I can run either npm run dev or npm run build, depending on what I want to do with my project.

## Installing and running a server with hot reload

When working with a tool like webpack, it’s good to mimic a real server environment locally rather than dealing with the file:// protocol. You might already have something set up for that (for example, I use [XAMPP](https://www.codeinwp.com/blog/local-wordpress-development-tools/) since I work a lot with PHP and [WordPress](https://www.codeinwp.com/blog/wordpress-statistics/)). But webpack gives you the option to easily install a server with live reloading, should you need it.

To install a server as a developer dependency, I’m going to run the following command in my project’s root directory:

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

Once that’s installed, I’m going to add a few lines to my webpack.config.js file:

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

const path = require('path');

module.exports = {

plugins: [

new HtmlWebpackPlugin({

hash: true,

title: 'Webpack Example App',

header: 'Webpack Example Title',

metaDesc: 'Webpack Example Description',

template: './src/index.html',

filename: 'index.html',

inject: 'body'

})

],

mode: 'development',

output: {

clean: true

},

devServer: {

contentBase: './dist',

open: true

}

};

Code language: JavaScript (javascript)

Notice I’ve added a section to module.exports called devServer. In there I’ve specified where I want my pages served from and I’m using the open option to open my project in a browser automatically. The browser serves the page using the default port of 8080 so I can view my page using the URL:http://localhost:8080/.

One final thing I need to do is add the server as part of my build script in package.json:

"scripts": {

"test": "echo \"Error: no test specified\" && exit 1",

"dev": "webpack serve --mode development",

"build": "webpack --mode production"

},

Code language: JavaScript (javascript)

Notice I’ve added serve to the dev script. I could do the same for the build script for production if I want. With this in place, I can run the following command to build my bundle and serve my pages from localhost:8080:

npm run dev

This opens the browser so I can view my page. This will live reload too, so if I make any changes to my content in the src folder, the bundle will be built again in dist and the browser will automatically reload the page.

## Final example of a webpack implementation

With all the above in place, the npm run dev command will produce my build each time it’s executed. I can then use the following command to build my project for production:

npm run build

This executes the build script in production mode (as outlined in my package.json). In my case, this produces a minified version of the following inside index.html in the dist folder:

<!doctype html>

<html lang="en">

<head>

<title>Webpack Example App</title>

<meta charset="UTF-8"/>

<meta name="viewport" content="width=device-width,initial-scale=1"/>

<meta name="description" content="Webpack Example Description"/>

</head>

<body>

<h1>Webpack Example Title</h1>

<div id="wrapper" style="height: 120px">

<div class="panel"></div>

<div class="panel"></div>

<div class="panel"></div>

</div>

<div id="zoom-scene"></div>

<script defer="defer" src="main.js?097d8b8eda8ecc97a023"></script>

</body>

</html>

Code language: HTML, XML (xml)

Webpack has updated my placeholders with the info generated by HtmlWebpackPlugin, the rest of my template remains intact, and — most importantly — webpack has bundled my JavaScript modules into a single main.js file. The file reference uses a query string value to ensure the browser loads the new version and not a cached version.

If I view my page in the browser, I’ll see the two console logs that confirm that webpack bundled my dependencies correctly. If you’ve followed along with this webpack tutorial, you should get the same result.

## Wrapping up this webpack tutorial

That should be enough to get you started with webpack!

I’ve covered quite a bit in this webpack tutorial for beginners but I’ve only scratched the surface of what’s possible. For example, I haven’t incorporated any features that utilize CSS or images. The [official webpack docs](https://webpack.js.org/concepts/) are pretty technically heavy, but you should be able to find a lot to expand on what I’ve presented in this tutorial, particularly for [front-end development](https://www.codeinwp.com/blog/best-front-end-tools/).

You’ll note that I didn’t do anything in this webpack tutorial with the two utilities I installed earlier. As mentioned, those were for demonstrating bundles. If I include some JavaScript to deal with those tools, webpack will bundle all of that in the same way as the two console logs I added.

If you follow the steps as I’ve outlined above, you shouldn’t run into any major problems. If you do, feel free to post them in the comments here or try to do an online search for whatever error message you’re getting. Webpack is pretty big so it’s likely you’re not the only one that’s had the problem in question!