Tutorial: how to build a Webpack app with Vanilla JS or React

Jeremy Gottfried Follow

Aug 9, 2018 · 6 min read

Webpack is a great tool for delivering JS and CSS efficiently in an app build. Most frontend frameworks have built in scripts to make it easier to minify/uglify your scripts. However, if you want to deploy an app without frontend frameworks like React or Angular or if you simply want more control over your minified build, this guide will demonstrate the basics of configuring and deploying Webpack in development and production builds. Here are the steps:

1. Build a basic app file structure

Here is what your app directory will look like:

```
my-webpack-app
├── README.md
├── node_modules
├── package.json
├── .gitignore
├── public
│   ├── favicon.ico
├── src
│   ├── index.css
│   ├── index.js
├── webpack.config.js
├── index.html
├── server.js
```

To create this tree, run the following in your terminal (copy/paste):

```
mkdir my-webpack-app && cd my-webpack-app && touch README.md && npm
init && git init && touch .gitignore && mkdir public && mkdir src &&
touch src/index.css && touch src/index.js && touch webpack.config.js
&& touch index.html && touch server.js
```

IMPORTANT: You will need to provide your own app icon. You can convert any image into a favicon.ico file here: https://www.favicon-generator.org/

After you generate your favicon.ico file, place it in the public directory.

Add the following to your .gitignore file:

```
#.gitignore

node_modules
npm-debug.log
```

2. Set up your index.html

If you don't already have your own `index.html` set up, you can copy/paste the following code to your `index.html` file:

```
<!-- index.html -->
```

```
<!DOCTYPE html>
<html lang="en" dir="ltr">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Sample Webpack App</title>
  </head>
  <body>

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

This html tells your app to load its scripts from `bundle.js` .

IMPORTANT: If you are using your own HTML, make sure you change your script tag to `<script src=”/bundle.js”></script>` .

3. Configure your Webpack

First, install some dependencies to make sure webpack works properly. In your terminal, run the following:

```
npm install -save webpack webpack-cli webpack-dev-server style-loader
css-loader script-loader
```

Webpack.config.js is where you will configure webpack. Open the file in your favorite text editor.

Copy/paste the following code:

```
// webpack.config.js

module.exports = {
  entry: [
    './src/index.js',
    './src/index.css'
  ],
  output: {
    path: __dirname,
    publicPath: '/',
    filename: 'bundle.js'
  },
  module: {
    rules: [
      {
        test: /\.js$/,
        exclude: /node_modules/,
        use: {
          loader: "script-loader"
        }
      },
      {
        test: /\.css$/,
        use: [
          {
            loader: "style-loader"
          },
          {
            loader: "css-loader",
            options: {
              modules: true,
              importLoaders: 1,
              localIdentName: "[name]_[local]_[hash:base64]",
              sourceMap: true,
              minimize: true
            }
          }
        ]
      }
    ]
  }
};
```

Let’s break this code up so you understand each component:

Establish entry points for JS and CSS:

```
  entry: [
      './src/index.js',
      './src/index.css'
    ]
```

The `entry` object tells webpack which files to convert into minified/uglified JS and CSS. You can add any files to the array that you want to include in the final build.

Establish output paths for your Webpack build:

```
output: {
    path: __dirname,
    publicPath: '/',
    filename: 'bundle.js'
  }
```

The `output` object establishes the paths where webpack will store its final build.

`filename` is the name of the file where the build will be stored.

`publicPath` is the public URL that the build will connect to. `/` tells webpack that this build will be the path to our app's root URL.

`path` is the absolute path, i.e. the full name of the directory where our build is stored. `__dirname` is a built in node.js variable that maps to the current directory. If the current file is `Users/jeremy/my-webpack-app/bundle.js`, `console.log(__dirname)` will print out `Users/jeremy/my-webpack-app`.

Establish rules for the modules in your app:

```
module: {
    rules: [
      {
        test: /\.js$/,
        exclude: /node_modules/,
        use: {
          loader: "script-loader"
        }
      },
      {
        test: /\.css$/,
        use: [
          {
            loader: "style-loader"
          },
          {
            loader: "css-loader",
            options: {
              modules: true,
              importLoaders: 1,
              localIdentName: "[name]_[local]_[hash:base64]",
              sourceMap: true,
              minimize: true
            }
          }
        ]
      }
    ]
  }
```

The `module` object is where we define rules for how webpack should handle various modules, such as JS and CSS. We are telling webpack to use `script-loader` to load JS, and `style-loader` or `css-loader` to load css.

There are many different webpack loaders to use for different types of app needs. For example, if you wrote your JS components in the babel format, you can use `babel-loader` instead of `script-loader` for your JS. See a full list of loaders here: https://webpack.js.org/loaders/

Remember that you need to add any webpack loaders you use to your npm dependencies.

4. Create an Express JS server to serve your build

In server.js, copy/paste the following code:

```
// server.js

const express = require('express');
const favicon = require('express-favicon');
const path = require('path');
const port = process.env.PORT || 8080;
const app = express();
app.use(favicon(__dirname + '/public/favicon.png'));
// the __dirname is the current directory from where the script is
running
app.use(express.static(__dirname));
```

```
// send the user to index html page inspite of the url
app.get('*', (req, res) => {
  res.sendFile(path.resolve(__dirname, 'index.html'));
});

app.listen(port);
```

This code creates a Node.js server to serve your webpack build. The server is set up to serve the development app to port 8080, but it will also work for serving a production build.

Right now, the server is set up to send any URL request to `index.html` . You can also replace the `*` with path names if you want to send different paths to different files.

5. Add some basic JS and CSS to test that Webpack is working

To see that the app is working, you can add some basic JS and CSS to your `index.js` and `index.css` files.

In index.js, you can use the following:

```
// src/index.js

document.addEventListener("DOMContentLoaded", function(event) {
  const element = document.createElement('h1')
  element.innerHTML = "Hello World"
  document.body.appendChild(element)
})
```

This is a basic script to create a new `h1` element and fill it with the text `Hello World`

In index.css, you can use the following:

```
/* src/index.css */

h1 {
  text-align: center;
}
```

6. Configure your package.json scripts

In order for webpack to work, we need to add some scripts to our `package.json` file.

```
//package.json

"test": "echo \"Error: no test specified\" && exit 1",
    "dev": "webpack-dev-server --mode development --open",
    "start": "webpack -p && node server.js"
```

These scripts set up your app for development and production. `webpack -p` creates a production build and `node server.js` starts your express server.

`webpack-dev-server` is a webpack library that can dynamically load changes to your code in a development environment, so you can view changes quickly.

To run the app in development, simply run the following in your terminal:

```
npm run dev
```

7. Using Webpack in React

If you want to use Webpack with React, you need to do a couple extra things.

Run the following in the terminal:

```
touch .babelrc
```

In `.babelrc` , copy/paste the following:

```
{ "presets": ["env", "react"]}
```

This tells our app to run babel in a React environment.

In `webpack.config.js` , change your JS loader to `babel-loader` :

```
//webpack.config.js

{
        test: /\.js$/,
        exclude: /node_modules/,
        use: {
          loader: "babel-loader"
        }
      }
```

Install a few extra dependencies:

```
npm install -save babel-core babel-loader babel-preset-env babel-
preset-react react react-dom
```

Change your index.js file to load a basic react app:

```
// src/index.js

import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';

ReactDOM.render(
  <App />, document.getElementById('root'));
```

Make sure to create your parent component in a file called `App.js`

Now you have all the tools to use Webpack in a React app or a Vanilla JS app. Enjoy!

8. Deploy a production build to Heroku

If you don’t already have a heroku account, create one here:
https://signup.heroku.com/

In your command line, run the following:

```
heroku login
```

You will need to type in your heroku credentials to the terminal. Once you have successfully entered your heroku credentials, run the following in your terminal to create a new deployed app:

```
heroku create sample-webpack-production-app
```

Replace sample-webpack-production-app with your own app name.

Then push your app build to heroku with the following git in your terminal:

```
npm install
git add .
git commit -m "initial commit"
git push heroku master
```

These commands install your dependencies and connect your repo to the remote repository hosted by heroku.

Congrats! Now you've completed all the necessary steps to deploy a Webpack build. To view your app, run the following in the terminal:

```
heroku open
```

Enjoy! You can also see what the final repo looks like here:

jeremygottfried/sample-webpack-app

sample-webpack-app — A sample production-ready app using webpack and vanilla JS

Webpack and Vanilla JS

github.com

JavaScript React Webpack Web Development Frontend