Lesson **Weekend**

Intermediate JavaScript (/intermediate-javascript) / Test-Driven Development and Environments with JavaScript (/intermediate-javascript/test-driven-development-and-environments-with-javascript) /

Configuring webpack and Using npm Scripts

Text

Before we get started with configuring webpack, let's take a look at our shape tracker application so far. It should have the following:

- The basic project files of html, JS, and CSS.
- A `.gitignore` file.
- A `package.json`, `package-lock.json`, and `node_modules`.
- The `package.json` should show two dev dependencies installed: webpack and webpack-cli.

If you haven't already created a repo on GitHub, do so now and push your local project to your remote repo.

Then, follow along with this lesson to create an npm script and `webpack.config.js` file in your project.

Creating an npm Script

Remember that `"scripts"` field in our `package.json`? It's time to add our first script. npm scripts let us use the `npm run [NAME-OF-SCRIPT]` command to run routine processes related to development. In many cases, we'll create a script that calls on a package that's been downloaded to our project.

Since we're using webpack to build our applications, we'll write a `build` script. What does it mean to build our code? Generally speaking, building a project's code means to compile it so that it is prepared to run. With webpack, it specifically means to take all the different files in our `src` directory, which are easy for humans to read and organize, and then bundle them into a single file that's much leaner and more efficient for browsers.

Let's get rid of the `test` script for now (we'll add one back later when we start writing tests). The `scripts` section of our `package.json` file should look like this:

package.json

```
...

"scripts": {
    "build": "webpack"
  },
...
```

Bundling JavaScript with webpack

Now that we have a `"build"` script defined, we can invoke it in our terminal:

```
$ npm run build
```

What this npm script does is find webpack amongst our project's dependencies, and then run webpack so it bundles our code.

When we do this, we should get output similar to what follows:

```
$ npm run build

> shape-tracker@1.0.0 build
> webpack

Hash: eefb8d2c5f73122ea6b3
Version: webpack 4.46.0
Time: 338ms
Built at: 06/01/2022 7:44:46 PM
  Asset      Size  Chunks             Chunk Names
main.js  1.31 KiB       0  [emitted]  main
Entrypoint main = main.js
[0] ./src/index.js 677 bytes {0} [built]

WARNING in configuration
The 'mode' option has not been set, webpack will fallback t
o 'production' for this value. Set 'mode' option to 'develo
pment' or 'production' to enable defaults for each environm
ent.
You can also set it to 'none' to disable any default behavi
or. Learn more: https://webpack.js.org/configuration/mode/
```

This output contains information about webpack's bundling process, including what files it has found to bundle (in this case just `./src/index.js`), and the name of the bundle it has created (`main.js`). If we look in our project's file tree, we'll see a new folder called `dist` with a file inside called `main.js`. And if we look inside `main.js`, we'll see minified JavaScript! This is the essence of what webpack does: bundle and optimize code.

Stop and take a look at the minified JavaScript in `main.js` now — we'll be comparing it to other code in just a second.

You might be thinking, great, this is webpack at work — so what? Well, what's notable about this is that webpack will bundle our JavaScript without any configuration as long as we have one JS file called `index.js`. That's because without a configuration file, webpack assumes our project's entry point is called `index.js` and the bundled code should be output to `dist/main.js`.

However, webpack has just bundled one single file in our project — `index.js`, but not `triangle.js`, and that's just not going to cut it for us. So, if we want webpack to include all of our JS files, and other dependencies, we're going to have to configure webpack and learn a new way of connecting files to each other.

As for the warning, let's resolve that next.

Development and Production Modes

The warning message is decently clear: we need to set whether we're building our code for development or production. We can set the mode by altering the npm script for `"build"` in `"scripts"`:s

package.json

```
"scripts": {
  "build": "webpack --mode=development"
},
```

As we can see, the `--mode` flag lets us specify a development environment. We could instead set `--mode=production` to specify a production environment, but we're in development, so we don't want that right now.

However, setting webpack to bundle code in development mode changes how webpack bundles our code. Let's see how this works. Run `$ npm run build` again in the command line. Now there's no warning and that's because we've set a value for mode.

Next, look inside `dist/main.js` — the bundled JavaScript now looks radically different! It has NOT been minified. Well, minification provides optimization benefits for production websites but it's not a priority for development, because we want to be able to look inside the bundle and debug it if necessary.

In the coming lessons, we'll revisit the topic of production versus development code.

Configuring webpack

We configure webpack through a configuration file. This file should go in the top level (or 'root') of our project directory. In order for webpack to find this configuration file, it must be named `webpack.config.js`. Let's create it now.

First create a new file called `webpack.config.js` in the root of the Shape Tracker directory. With this addition, your Shape Tracker project structure should look like this:

```
shape-tracker/
├── dist
│   └── main.js
├── node_modules
├── index.html
├── src
│   ├── index.js
│   └── triangle.js
├── css
│   └── styles.css
├── .gitignore
├── package-lock.json
├── package.json
└── webpack.config.js
```

Then, add the following contents to `webpack.config.js`:

webpack.config.js

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

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

This is a very basic configuration, and we'll continue to add to it in coming lessons. The syntax may seem confusing at first, but we can break this down a bit. For instance, we can see that `module.exports` is a property that holds an object with key-value pairs. When we come across code that looks different, don't forget that we can always apply our basic understanding of JavaScript to make it a bit less overwhelming.

When webpack creates the bundled code, it will use the instructions in `webpack.config.js` to do so. This is just like how npm uses the instructions in `package.json` to install a project's dependencies.

Let's go over the main points of this new code:

1. Anything webpack requires to run goes at the top of our configuration file. For now, we only have `require('path')`, which we add to configure the path where our output files will be saved. We save this in a `const` variable. We'll soon be adding more `require` statements, which we will cover further in the next lesson.
2. We specify an **entry point**. This is the JS file where webpack will enter our application and then use a dependency graph to load all other required JS files. That just means webpack is going to gather all the files that our entry point depends on, then gather all the files that those files depend on, and so on

— until all the dependencies have been gathered. As we can see, webpack will enter through the file where we store our user interface logic: `'./src/index.js'` . The file name `index.js` is a default entry point for webpack.
3. We also specify an **output**. This is where all our code will go after it's been processed and bundled. Instead of relying on the default file name of `main.js` , we'll use `bundle.js` . Every time we run `$ npm run build` , webpack will follow the instructions in `webpack.config.js` and create a `dist/bundle.js` file.

Note that we only have access to `path.resolve()` because we required it at the top of our configuration file. `path` is a part of webpack's internal tooling.

In short, the instructions above tell webpack to go into our entry point file `src/index.js` , gather all its dependencies (and its dependencies' dependencies...), "webpackify" them (concatenate them into one file), call the concatenated file `bundle.js` and store them in the `dist` directory.

Now that we have a very basic configuration file in place, we're almost ready to process and concatenate some JavaScript code. Next, we need to learn about `import` and `export` statements in order to connect our files together.

Previous (/intermediate-javascript/test-driven-development-and-environments-with-javascript/introduction-to-webpack)
Next (/intermediate-javascript/test-driven-development-and-environments-with-javascript/es6-imports-and-exports)
Lesson 10 of 49
Last updated more than 3 months ago.

disable dark mode

Epicodus (http://www.epicodus.com)