Lesson Weekend

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

Text

Let's update our shape tracker application to use `import` and `export` statements. By the end of this lesson, we will be using webpack to bundle all of our JavaScript files for us.

Remember, bundling JavaScript is webpack's core functionality. Bundling other modules like CSS, images, and other assets is made possible through loaders, which we'll begin to incorporate in the next lesson!

Bundling JavaScript

First, let's start by adding a `dist` directory. Next, move `index.html` inside the `dist` directory. If you already have a `dist` directory in your project, simply move `index.html` inside of it.

Locating your HTML file within the `dist` directory is the basic setup for a project that uses webpack (https://webpack.js.org/guides/getting-started/#basic-setup). However, for us this will be a temporary change. We will be moving `index.html` back to the `src` directory in a few lessons when we implement a plugin that directs webpack to create an output HTML file based on a template we provide.

By the way, don't forget that we've added `dist/` to our `.gitignore` file. If we were to push our updated code, changes to `index.html` wouldn't get pushed. This is fine because this change is temporary — and there is no need to commit our code just yet. However, if you did want to explore using a basic webpack setup in the future, you'd want to update your `.gitignore` file to only ignore `dist/bundle.js` and not the entire directory and HTML.

Here's our current project structure:

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

Note that we aren't including automatically generated files and directories like `node_modules` and `package-lock.json` in the structure above. We're only including the files and directories that we need to add manually.

Adding `import` and `export` Statements

Next, let's update our JavaScript files to use `import` and `export` statements. We'll use a default export for `Triangle` because it's the only thing we'll export from the file.

src/triangle.js

```
export default function Triangle(side1, side2, side3) {
  ...
}

...
```

Now we need to make sure we import our `Triangle` constructor at the top of `index.js`:

src/index.js

```
import Triangle from './triangle.js';
...
```

Updating `index.html`

Finally, we'll make some small updates to the `<head>` of `index.html`:

```
...
<head>
  <script type="text/javascript" src="bundle.js"></script>
  <link rel="stylesheet" href="./../css/styles.css">
  <title>Shape Tracker</title>
</head>
...
```

Now we only have one JavaScript tag — for `bundle.js`. That file doesn't exist yet, and that's because webpack creates it for us.

We've also updated the **relative path** to our stylesheet to `./../css/styles.css`. That's because we've moved `index.html` to our `dist` folder, and we now have to go out of the `dist` folder in order to get to the `css` directory to access `styles.css`. Take note of a few things:

- Two dots `/../` in a path indicates that we are moving up a level from the directory we are in, while one dot `./` in path indicates our current working directory.
- So the full path `./../css/styles.css` says, to load the CSS in `styles.css`:
  - Start in this directory `./`
  - Then move up a directory `/../` — this is the project's root
  - Then go into the `/css/` directory
  - Then locate `styles.css`

Note that this path will change again in the next lesson when we use webpack to bundle our CSS.

Running webpack

Now run `$ npm run build` in the root directory of the project to invoke webpack to do its job and bundle our JavaScript files.

webpack will start by accessing the entry point at `src/index.js`. Then, webpack will recursively add any dependencies (anything that needs to be imported from elsewhere). Since `index.js` only imports `Triangle` from `triangle.js` and `triangle.js` has no imports of its own, our project currently only has one dependency.

If all goes well, your output will look something like this:

```
> shape-tracker@1.0.0 build /Users/staff/Desktop/shape-trac
ker
> webpack

Hash: 7d88ba320f665950d074
Version: webpack 4.46.0
Time: 70ms
Built at: 06/01/2022 9:33:20 AM
    Asset      Size  Chunks             Chunk Names
bundle.js  1.27 KiB       0  [emitted]  main
Entrypoint main = bundle.js
[0] ./src/index.js + 1 modules 650 bytes {0} [built]
    | ./src/index.js 443 bytes [built]
    | ./src/triangle.js 207 bytes [built]
```

Note: If you get errors, you probably missed a step from previous lessons. Make sure you've actually added a `webpack.config.js` file with a configuration and updated the scripts section of `package.json` to include `"build": "webpack --mode=development"`.

Just like with before, we we'll see that webpack creates a `bundle.js` file in the `dist` folder. If we look inside of it, we'll see the code from both `index.js` and `triangle.js`. This means that webpack has done it's job to concatenate our files together.

The code inside of `bundle.js` is NOT minified. That's because we've configured webpack to build its bundle in development mode. Minifying code is not as important for development as being able to look through the bundle and debug any code within. We'll revisit the topic of debugging in an upcoming lesson.

Now we can open our `index.html` file in the browser and our code will work. Woo-hoo!

One important detail to note is that if we go to the console and check the `window` object, we'll no longer see `Triangle` just hanging out in the global scope. We can also verify this by typing in the

following:

```
> window.Triangle;
undefined
```

Our JavaScript code is now modularized and code is scoped where it is needed. This is particularly important as our projects get larger.

So far, we haven't reduced our code much — all we've done is remove a single script tag from our code. However, it's a big deal when a project has hundreds of dependencies. More importantly, our JavaScript logic is separated in different files — which is great for human organization — but then bundled into one file — which is great for speed and efficiency.

We've barely scratched the surface of what webpack can do. In the next several lessons, we'll bundle our CSS, customize webpack for linting, and set up a live development server.

Previous (/intermediate-javascript/test-driven-development-and-environments-with-javascript/es6-imports-and-exports)
Next (/intermediate-javascript/test-driven-development-and-environments-with-javascript/bundling-css-with-webpack-loaders)

disable dark mode

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

© 2023 Epicodus (http://www.epicodus.com/), Inc.