Lesson | Weekend

# Intermediate JavaScript (/intermediate-javascript) / Test-Driven Development and Environments with JavaScript (/intermediate-javascript/test-driven-development-and-environments-with-javascript) / Processing HTML with a webpack Plugin

Text

So far we've used webpack to bundle our JavaScript and CSS files. We can also use webpack to generate HTML files for us. Note that we're not using webpack to bundle HTML with our JS and CSS — instead, we're using webpack to do some additional work for us by having webpack process and generate our HTML based on a template we provide to webpack.

Why would we do this? A bigger application could have many HTML files, all with multiple different entry points. In fact, this is a feature of webpack — to have multiple entry points instead of just one. With multiple entry points, we can tell webpack to create a bundle with dependencies for each entry point we specify. In turn, this gives us flexibility to load just one or all bundles, or load them at different times. This is a more advanced topic that you don't need to understand right now. At Epicodus, we'll only use one entry point with one bundle.

Let's get back to why we would use webpack to generate our HTML. Considering again an application with multiple entry points, what if someone adds an entry point or changes the `output` configuration in `webpack.config.js`? Our HTML file's script tags would then also need to be updated, too. That might lead to errors, and it is something we would need to manage carefully in order to avoid errors.

In an ideal world, we should only have to update our configuration file, and have our changes automatically be applied by webpack. Well, that's just what we'll do with a webpack plugin called HtmlWebpackPlugin (https://webpack.js.org/plugins/html-webpack-plugin/) — we'll let webpack do the heavy lifting of processing and generating our HTML files.

By the end of this lesson, you should have HtmlWebpackPlugin installed and configured in your Shape Tracker project.

## webpack Plugins

To add this functionality to webpack, we'll use our first **plugin** called HtmlWebpackPlugin.

What's the difference between a webpack loader and a webpack plugin?

- Loaders preprocess code that webpack can't directly work with, which means any file that's not JavaScript. Generally loaders process code before or during the creation of our bundle, and they only work with specific file types.
- Plugins are more powerful. They can modify and work with the entire bundle, so they generally run after the bundle has been created.

If the difference between loaders and plugins is feeling fuzzy, that's nothing to be worried about. You can continue using webpack without knowing the difference between loaders and plugins. Also,

don't worry about spending extra time trying to understand the difference between the two. For the most part, once we have an environment set up, we can forget about it — at least until we need to add or update new packages.

## Using HtmlWebpackPlugin

Before we install the plugin, **let's move our `index.html` file from the `dist` folder back into the `src` folder**. Why? From now on, we'll let webpack handle generating our HTML and outputting it to `dist`, and the `src/index.html` will be the HTML template that webpack uses to do this.

This is why `dist/` is in our `.gitignore` file — webpack will automatically generate everything inside this directory for us. Any developer can clone our project and then build it from our source code — so it would be redundant to push that code to GitHub.

Now let's add another dev dependency with npm. In the terminal, navigate to the root of the Shape Tracker project and enter the following command.

```
$ npm install html-webpack-plugin@4.5.2 --save-dev
```

Let's also update `webpack.config.js` to use this new plugin:

**webpack.config.js**

```
const path = require('path');
// There's a new line below this one!
const HtmlWebpackPlugin = require('html-webpack-plugin');

module.exports = {
  entry: './src/index.js',
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist')
  },
  // The plugins key below this line is also new!
  plugins: [
    new HtmlWebpackPlugin({
      title: 'Shape Tracker',
      template: './src/index.html',
      inject: 'body'
    })
  ],
  module: {
    rules: [
      {
        test: /\.css$/,
        use: [
          'style-loader',
          'css-loader'
        ]
      }
    ]
  }
};
```

- First we require `html-webpack-plugin` and make it available in `webpack.config.js`.
- Next we add a new key to the `module.exports` object called `plugins`.
- Finally, we instantiate a new `HtmlWebpackPlugin`. Our instantiated plugin is taking three arguments:

- `title` : This will be the title of our auto-generated HTML file.
- `template` : This is the HTML file we'll use as a template. Here we specify that it should be the `index.html` file we just moved to the `src` folder. If we didn't specify a template file, then webpack would just generate a file with a `<head>` and `<title>` and `<script>` tags.
- `inject` : This is a nice little option. webpack will inject our script at the bottom of our HTML for us. It's yucky to put script tags in the body when we're writing code because it's hard to read and mixes HTML and JS, which are separate concerns. However, our code will be more performant in production if we put our script tags there. webpack gives us the best of both worlds. We can write clean code when we're developing and then let webpack make it more performant for us by moving the script tags to the bottom of the HTML.

Also, because all the code in our HTML file loads synchronously, line by line, putting our script tags at the end of our HTML file means that we NO longer need to create an event listener for the `window` 'load' event in our code:

```
window.addEventListener('load', function() {
  // We set up event listeners here.
});
```

That's because we use it to prevent our scripts from running before our HTML has loaded so that we can ensure that our HTML elements exist and we can target them with document query methods. Moving our script tag to the bottom of our HTML file will reorder the loading process of our project's dependencies: first our HTML will load, then our scripts will load. Again, this happens because our script tag has been added to the bottom of our HTML body.

All that said, we won't remove the event listener for the `window` 'load' event from our example projects. However, you can if you would like to, as long as your scripts tags are at the bottom of your HTML's body tag, with or without using webpack.

**Note:** Some students have found that their projects load faster *in development* when they use `inject: 'head'` instead of `inject: 'body'`. Ultimately, you may use either option based on your preferences. However, keep in mind that having the script at the end of the body is preferable for production sites.

Before we `$ npm run build` our application again, let's remove the `<script>` tag currently linking our bundled JavaScript from `index.html`. Here's the updated `<head>`:

**src/index.html**

```
...
<head>
  <title>Shape Tracker</title>
</head>
...
```

Now run `$ npm run build`. webpack will add `index.html` to our `dist` folder for us. If you take a look, you'll see that webpack has added a script tag for our bundled JavaScript to the bottom of our HTML file.

At this point, we can open `dist/index.html` (either by opening it in the browser or by using Live Server), and our site will be there. Everything appears exactly the same — but our code is now bundled.

Previous (/intermediate-javascript/test-driven-development-and-environments-with-javascript/bundling-css-with-webpack-loaders)
Next (/intermediate-javascript/test-driven-development-and-environments-with-javascript/improving-development-by-automating-

clean-up-tasks)

Lesson 14 of 49
Last updated more than 3 months ago.

disable dark mode


(http://www.epicodus.com)