Transpiling

Webpack

Now that we can download dependencies into our project, we'll learn how to use Webpack to bundle our project for the web. Once we're familiar with this, we'll be able to transpile projects written in ES6 so that they work in the majority of browsers.

Webpack is a "module bundler". It can be used to convert files into different formats, and combine multiple script files into either a single bundle or performant "chunks" of code for usage on the web. It's not exclusively for Javascript either: Webpack can convert images into base64, SASS into CSS, and much more.

In this session, we'll go over how to combine two script files into one for the web.

First I'll explain how to include scripts in other scripts in NodeJS.

I'll create a file that defines a function that outputs the input number multiplied by itself:

```
var multiply = function (number) {
  return number * number
}
```

To make this function visible to other files, we'll need to export it. There are a number of ways to do this, but the approach you'll most commonly see in NodeJS is the "CommonJS" approach. One way is to define a `module.exports` value:

```
var multiply = function (number) {
  return number * number
}

module.exports = multiply
```

Now, we'll create another file, and import this dependency:

```
var multiply = require('./multiply.js')

console.log(multiply(3))
```

As we can see here, to require a local dependency, specify a relative path, and include .js. Whatever is exposed via `module.exports` will be available when the file is `require`'d.

When the `index.js` file is run, the multiply function will be included and called from the `index.js` script.

So how can we do this in the browser?

Well, let's try. Create an index.html file, and include the index.js file as a script:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Webpack Bundle</title>
  </head>
  <body>
    <script src="./index.js"></script>
  </body>
</html>
```

When you run this, you'll see a `require is not defined` error in the console. And it's true: Javascript in the browser does not have a global `require` function.

Now, it is true that ES6 introduces an alternative way to export and import sections of scripts as well, and at the time of preparing this tutorial, NodeJS supports this method as an experimental feature. I've included a sample of exporting/importing in ES6 in the sample code in case you're curious to see what that looks like.

In our case, we want to combine the script into one file to avoid an extra http call, and later, we'll want to be able to transform the script files into a more compatible version of Javascript. So we will not be using ES6 modules in this session.

To combine everything into one bundle, let's turn this folder into a Node project. Running `npm init` and using all the default options should be sufficient.

```
npm init
```

Should generate:

```
{
  "name": "webpack-sample",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

Now, we'll need to install Webpack. We'll specify a dependency and version in package.json as opposed to the `npm install` command. That way, we can define a specific version. Different versions of Webpack require different configurations.

```
  "dependencies": {
    "webpack": "4.29.6"
  },
```

Once the dependency is specified, install it by running `npm install`. This will download the dependency into the created node_modules folder and create a package-lock.json file.

Now, we need to configure Webpack and tell it to generate a combined script file for the web based off the `index.js` file we created earlier. There are a number of different ways to define a Webpack configuration. I find the most versatile way to use Webpack is to create a script file and invoke it from there.

To that end, let's create a script that we can run from npm:

(build.js):

```
console.log('Called build script')
```

And then update the package.json file:

```
{
  "name": "webpack-sample",
  "version": "1.0.0",
  "description": "",
```

```
  "main": "index.js",
  "scripts": {
    "build": "node build.js",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "dependencies": {
    "webpack": "4.29.6"
  },
  "author": "",
  "license": "ISC"
}
```

Now we'll make sure it works:

```
npm run build
```

Now to call the Webpack build function from the build script. I'll walk through what each parameter does as it's added. A more complete guide can be found on the webpack website.

```
var webpack = require('webpack')
var path = require('path')

webpack({
  entry: './index.js',
  mode: 'development',
  output: {
    path: path.resolve('./'),
    filename: 'bundle.js'
  }
}, function (error, stats) {
  if (error) {
    console.error(error)
  } else {
    console.log(stats.toString())
  }
})
```

We'll include `webpack` and `path` in this script. We need `path` to get the absolute path for one of the parameters.

One of the ways to use Webpack is to call the imported code as a function. The first argument is the configuration object, and the second argument is the callback function to invoke once webpack either completes or throws an error.

The `entry` is the relative path (or relative paths) to the "main" script file. As the `index.js` file is the "top level" script, we'll define this as the entry point.

`mode` is an optional parameter that tells webpack how to apply certain optimizations. Options include `"production"` (default), `"development"`, and `"none"`. We'll use `"development"` for now.

The `output` object defines how webpack will generate its results. The path option is an absolute path that specifies where the outputted script should end up. The `filename` specifies what the name of the file will be.

The second argument to the webpack function is the callback. The first argument is null, unless any errors occur. Once webpack completes, details about the bundle constructed are returned in this function's second argument.

With this configuration, running the build script will generate a bundle.js file. We can often check to see if the bundle was created successfully by scrolling to the bottom of this file and ensuring something like the code we wrote is there:

```
/***/ "./index.js":
/*!******************!*\
  !*** ./index.js ***!
  \******************/
/*! no static exports found */
/***/ (function(module, exports, __webpack_require__) {

eval("var multiply = __webpack_require__(/*! ./multiply.js */
\"./multiply.js\")\n\nconsole.log(multiply(3))\n\n\n//#
sourceURL=webpack:///./index.js?");

/***/ }),

/***/ "./multiply.js":
/*!*********************!*\
  !*** ./multiply.js ***!
  \*********************/
/*! no static exports found */
/***/ (function(module, exports) {

eval("var multiply = function (number) {\n  return number *
number\n}\n\nmodule.exports = multiply\n\n\n//#
sourceURL=webpack:///./multiply.js?");

/***/ })
```

It's a little strange looking, but you can see the multiply function being called in the `index.js` section and defined in the `multiply.js` section.

You should get in the habit of checking that the outputted file generates the correct result, and also that it is in fact generated or updated upon running the build script. I've found that Webpack 4 is not particularly good about making errors obvious.

Now, just change index.html to look at the bundle:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Webpack Bundle</title>
  </head>
  <body>
    <script src="./bundle.js"></script>
  </body>
</html>
```

And then you should see 9 in the console:

```
9
```

It's great that Webpack can combine multiple files into one, but what kind of performance boost is it if the result is a file many times the size of the original files combined? Fortunately, Webpack has the ability to minify or condense the built script to use as little code as possible. This setting is on by default, but was disabled when the "development" mode was specified. If that option is disabled in the build script:

```
var webpack = require('webpack')
var path = require('path')

webpack({
  entry: './index.js',
  // mode: 'development',
  output: {
    path: path.resolve('./'),
    filename: 'bundle.js'
  }
}, function (error, stats) {
  if (error) {
    console.error(error)
  } else {
    console.log(stats.toString())
  }
})
```

You'll see that the resulting bundle is squished on to one line:

```
!function(e){var t={};function n(r){if(t[r])return t[r].exports;var o=t[r]=
{i:r,l:!1,exports:{}};return
e[r].call(o.exports,o,o.exports,n),o.l=!0,o.exports}n.m=e,n.c=t,n.d=function(e,t,r)
{n.o(e,t)||Object.defineProperty(e,t,{enumerable:!0,get:r})},n.r=function(e)
{"undefined"!=typeof
Symbol&&Symbol.toStringTag&&Object.defineProperty(e,Symbol.toStringTag,
{value:"Module"}),Object.defineProperty(e,"__esModule",
{value:!0})},n.t=function(e,t){if(1&t&&(e=n(e)),8&t)return
e;if(4&t&&"object"==typeof e&&e&&e.__esModule)return e;var
r=Object.create(null);if(n.r(r),Object.defineProperty(r,"default",
{enumerable:!0,value:e}),2&t&&"string"!=typeof e)for(var o in e)n.d(r,o,function(t)
{return e[t]}.bind(null,o));return r},n.n=function(e){var t=e&&e.__esModule?
function(){return e.default}:function(){return e};return
n.d(t,"a",t),t},n.o=function(e,t){return
Object.prototype.hasOwnProperty.call(e,t)},n.p="",n(n.s=0)}([function(e,t,n){var
r=n(1);console.log(r(3))},function(e,t){e.exports=function(e){return e*e}}]);
```

In this particular case, it's still a little longer than the scripts on their own, but in larger projects, this can take off a good 80% of the code. Obviously, minified code is harder to debug. There are some tools for working with minified code though, like sourcemaps, that we may get into in a future session.

Before we finish up, I wanted to show another way webpack can be used, as you may see this in the wild. The webpack site recommends specifying a configuration by writing a `webpack.config.js` file, which instead of calling webpack as a required module, expects the configuration to be exported, like this:

`webpack.config.js`:

```
var path = require('path')

module.exports = {
  entry: './index.js',
  // mode: 'development',
  output: {
    path: path.resolve('./'),
    filename: 'bundle.js'
  }
}
```

Then, invoking webpack from the command line:

`package.json`:

```
{
  "name": "webpack-sample",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "build": "node build.js",
    "build-config": "webpack",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "dependencies": {
    "webpack": "4.29.6"
  },
  "author": "",
  "license": "ISC"
}
```

```
npm run build-config
```

This will ask us if we want to install webpack-cli. Type "yes" and press enter.

And the same output is generated.

The downside of this approach is that it doesn't allow for combining multiple build steps later, and it can spread out the script building code into more files than are arguably necessary. You might want to do additional tasks when building a bundle, such as running tests or copying files to an output folder.

In conclusion, Webpack is a module bundler tool that can be used to combine and transform multiple Javascript files into a single bundle. It permits Javascript targeted for the front end to use CommonJS-style `module.export` and `require` to export and include files. And Webpack is extremely configurable.

Now that we know the basics of Webpack, we can make use of Babel, and finally move forward with ES6.