## **ES6 MODULES**

### **Introduction**

Separate from the module pattern that we discussed in an earlier lesson, “modules” is a feature that arrived with ES6. Browser support for this feature is quite slim at this point, but is slowly improving and until all modern browsers support it, we can make it work using an external module bundler. ES6 modules are starting to appear in many code bases around the net and getting them up and running will give us a chance to explore some new parts of the JavaScript ecosystem, so it’s going to be a worthy excursion!

Don’t be fooled! We’re going to cover much more than just the new module syntax in this lesson! Before we can really *use* these modules, we’re going to have to learn about npm and webpack which are both topics that will be *very* useful to you even beyond this lesson. In the end, the modules themselves are simple to implement, so we’re going to take this chance to learn about a few other things.

### **Learning Outcomes**

After completing this lesson, you will be able to:

* Explain what npm is and where it was commonly used before being adopted on the frontend.
* Describe what npm init does and what package.json is.
* Know how to install packages using npm.
* Describe what a JavaScript module bundler like webpack is.
* Explain what the concepts “entry” and “output” mean as relates to webpack.
* Briefly explain what a development dependency is.
* Explain what “transpiling code” means and how it relates to frontend development.
* Briefly describe what a task runner is and how it’s used in frontend development.
* Describe how to write an npm automation script.
* Explain one of the main benefits of writing code in modules.
* Explain “named exports” and “default exports”.

### **The History of JavaScript**

Why do we even need or want this stuff? What do you gain from all of this added complexity? These are good questions.. with good answers.

* Read [this article](https://peterxjang.com/blog/modern-javascript-explained-for-dinosaurs.html) for a bit of a history lesson. It’s long, but it puts what we’re doing here in great perspective. This article is a bit older, and those who have coded along with the example have frequently run into issues, so we don’t suggest that you code along (you’ll be following along with the offical Webpack documentation later). Nevertheless, this article is extremely important conceptually and really clarifies the ‘WHY’ of the rest of this lesson.

### **npm**

The node package manager is a command line tool that gives you access to a gigantic repository of plugins, libraries and tools. If you have done our Fundamentals course, you will probably have encountered it when you installed the Jasmine testing framework to do our exercises.

1. Take a couple minutes to read the [About npm](https://docs.npmjs.com/getting-started/what-is-npm) page - a great introduction to npm.
2. [This video](https://docs.npmjs.com/getting-started/installing-npm-packages-locally) teaches you how to install packages with npm.
3. [This tutorial](https://docs.npmjs.com/getting-started/using-a-package.json) covers the package.json file, which you can use to manage your project’s dependencies
4. If you run into trouble at any point you can check out [the official docs page](https://docs.npmjs.com/) for more tutorials and documentation.

### **Yarn?**

At some point, you will probably run into [Yarn](https://yarnpkg.com/en/) - a replacement for the default npm. For the most part, it does the same things though it *does* have a few more features. Recent versions of npm have incorporated some of the best features of Yarn, so using it won’t offer you any real advantages at this point in your career. It *is* a fine project, however, and may be worth your consideration in the future.

### **Webpack**

Webpack is simply a tool for bundling modules. There is a lot of talk across the net about how difficult and complex it is to set up and use, but at the moment our needs are few and the setup is simple enough. In fact, you can see an example of getting it up and running on the front page of [their website](https://webpack.js.org/).

Webpack *is* a very powerful tool, and with that power comes a decent amount of complexity - just look at the sample config file on [this page](https://webpack.js.org/configuration/) 😱. Don’t let it scare you off! The basic configuration is not difficult and proficiency with webpack looks *amazing* on resumes.

To get us started we are going to refer to the official documentation.

1. Code along with the first four steps of [this tutorial](https://webpack.js.org/guides/getting-started/) (“Basic Setup” through “Using a Configuration”)

Let’s discuss what’s going on there. After installing webpack using npm we set up a simple project that required an external library (lodash - check it out [here](https://lodash.com/) if it’s new to you) using a simple script tag. The site lists a few reasons why this is probably *not* ideal and then steps through using webpack to accomplish the same thing.

There are a couple of key concepts to understanding how webpack works - entry and output. In this example, we rearranged the files into a src and dist folder. Technically we could have called those folders anything, but those names are typical. src is our *source* directory. In other words, src is where we write all of the code that webpack is going to bundle up for us. When webpack runs, it goes through all of our files looking for any import statements and then compiles *all* of the code we need to run our site into a single file inside of the dist folder (short for *distribution*). Our entry file, then is the main application file that links (either directly or indirectly) to all of the other modules in our project. In this example, it is /src/index.js. The output file is the compiled version - dist/main.js.

* browse [this document](https://webpack.js.org/concepts/) for more details. We’ll talk about plugins and loaders in another lesson.

### **ES6 Modules (finally!)**

Now that we (sorta) understand what webpack is doing it’s time to discuss the module syntax. There are only 2 components to it - import and export.

* Take a moment to look at the docs for [import](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/import) and [export](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/export).

Of course, the import statement is the same thing that you used during the webpack tutorial! These things are simple to use.

// a file called functionOne.js const functionOne = () => console.log('FUNCTION ONE!') export { functionOne }

// another JS file import { functionOne } from './functionOne' functionOne() //this should work as expected!

There are *many* benefits to writing your code in modules. One of the most compelling is code reuse. If, for instance, you have written some functions that manipulate the DOM in a specific way, putting all of those into their own file as a ‘module’ means that you can copy that file and re-use it very easily!

Other benefits include all of the benefits to wrapping your code in factory functions or using the module pattern (the module pattern and ES6 modules are not the same things.. this naming convention is frustrating). By using ES6 modules you can keep different parts of your code cleanly separated, which makes writing and maintaining your code much easier and less error-prone. Keep in mind that you can *definitely* export constructors, classes and factory functions from your modules.

To pull it all together, let’s write a simple module and then include it in our code. We are going to continue from where the webpack tutorial left off. Before beginning your file directory should look something like this:

├── dist

│ ├── main.js

│ └── index.html

├── src

│ └── index.js

├── package-lock.json

├── package.json

└── webpack.config.js

and you should be able to bundle and run webpack by simply typing npx webpack in the terminal.

Add a new file to the src directory called myName.js with the following contents:

const myName = (name) => 'Hi! My name is ' + name; export default myName

and then in src/index.js import and use your new function.

// import your function import myName from './myName'; function component() { const element = document.createElement('div'); // use your function! element.innerHTML = myName('Cody'); return element; } document.body.appendChild(component());

Easy! Now, if you run npx webpack in your project directory your page should show our new function being used.

There are 2 different ways to use exports in your code: named exports and default exports. Which option you use depends on what you’re exporting. As a general rule if you want to export multiple functions use named exports with this pattern:

const functionOne = () => 'ONE' const functionTwo = () => 'TWO' export { functionOne, functionTwo }

and to import them:

import {functionOne, functionTwo} from './myModule'

Using this pattern gives you the freedom to only import the functions you need in the various files of your program. So it’s perfectly fine to only import functionOne if that’s the only one you need.

The various import/export methods are best explained in the docs that we linked earlier - [import](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/import) and [export](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/export).