Lesson | Weekend

Intermediate JavaScript (/intermediate-javascript) / Test-Driven Development and Environments with JavaScript (/intermediate-javascript/test-driven-development-and-environments-with-javascript) / ES6 Imports and Exports

Text

When we build our code, how does webpack know which files are dependencies? Going back to our example of the peanut butter and jelly sandwich example, how do we let webpack know that `jelly.js` needs `blueberry.js` to work properly?

While we previously used our HTML file to connect and load all of our project's dependencies (JS, CSS, and any other files we might need for our project to run), this process doesn't work for webpack. Instead, webpack works directly with our JS, CSS, and other files through a specified entry point to bundle them into one package. Then, we write a `<script>` tag in our HTML that connects to this bundle. We haven't done this yet, but we will soon!

The takeaway here is that we can't rely on our HTML to connect and load our project's dependencies. Instead, we need to be able to connect them directly to one another, so webpack can handle loading and bundling them.

To do this, we need to use `import` and `export` statements in our code. These statements are a nice piece of functionality that was added to EcmaScript6 (the technical name for version 6 of JavaScript), which came out in 2015. In this lesson, we'll learn how to use these statements. We'll also spend time learning about an earlier way of achieving the same functionality with `require` statements and module exports. Then, in the next lesson we'll update the Shape Tracker project to use modern JavaScript's `import` and `export` statements so that webpack can bundle our JavaScript files.

`require` Statements and Module Exports

Before ES6, developers used NodeJS `require` statements to share logic between files. We've already seen a `require` statement in our `webpack.config.js` file in the following line:

webpack.config.js

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

...
```

Now let's take a closer look and work through some examples to understand what `require` statements and module exports do. **Do not add any of the following examples to your own code.** We're working through the following examples to understand the NodeJS way of connecting files.

Let's say we want to include our `Triangle` constructor in our user interface logic (`index.js`). First, we'd create a module export.

triangle.js

```
function Triangle(side1, side2, side3) {
  this.side1 = side1;
  this.side2 = side2;
  this.side3 = side3;
}

Triangle.prototype.checkType = function() {
  return "I can't answer that yet!";
};

// The following code is new.
// Don't add this to your own code.
exports.triangleModule = Triangle;
```

Let's look at the final line of code in this snippet. We take the `Triangle` constructor (on the right side) and assign it as a property to the `exports` object, calling it `triangleModule`.

Now we can import the `Triangle` constructor into our user interface logic like this:

index.js

```
const Triangle = require('./triangle.js').triangleModule;
```

We use the `require` statement to get the `triangleModule` from the specified relative path. Note that the relative path will vary depending on the project. In our project, `index.js` and `triangle.js` are in the same directory so the relative path is `./`. The `triangleModule` is saved in a variable so it can be used in this file.

With these changes, when we tell webpack to bundle our code with `$ npm run build`, webpack will be able to locate the `triangle.js` as a dependency of `index.js`, and both files will be added to the bundle.

Also, in `index.js` when we write the following code, invoking the `Triangle` constructor:

```
const triangle = new Triangle(3,3,3);
```

Here, `Triangle` actually refers to our `triangleModule` — *not* the `Triangle` constructor itself. However, since the `triangleModule` has been set to the `Triangle` constructor, they are essentially the same thing.

Importing and Exporting Code

Why bother to learn about `require` statements when we aren't going to use them in our own code? There are two reasons.

- You will see `require` statements a lot in other code, especially server-side Node code. They are all over the place and you need to recognize them and understand how they work.
- `import` and `export` statements, which we will be using, are just syntactic sugar for `require` statements. They are just using `require` under the hood.

Here's how we can use `import` and `export` instead. **Note: even though we will be using this in our code, you don't need to add it now — this lesson is for demonstration purposes.**

triangle.js

```
export function Triangle(side1, side2, side3) {
  this.side1 = side1;
  this.side2 = side2;
  this.side3 = side3;
}

Triangle.prototype.checkType = function() {
  return "I can't answer that yet!";
};
```

index.js

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

We specify that we want to `export` the `Triangle` constructor in our business logic file. Then we specify that we want to `import` the constructor in our user interface logic file. As we can see, it's a little bit cleaner than using `require` statements.

We can also have multiple export statements in a single file like this:

shapes.js

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

export function Circle(radius) {
  ...
}

export function Rectangle(side1, side2) {
  ...
}
```

Then we'd import all the shapes like this:

index.js

```
import { Triangle, Rectangle, Circle } from './shapes.js';
```

Note that in our own code, we will separate shapes out into their own files. However, there are plenty of situations where you might be exporting multiple things from a file (such as multiple different functions, whether a constructor for an object type, or just a function).

The exports in the examples above are called **named exports**. This is because the name of the thing being exported ***must*** **match** the name of the thing being imported. We can't say: `import { Thingy } from './../js/shape.js';` and expect JavaScript to know we mean `Triangle`.

Default Exports

If we only plan to export one thing from a file, we can use a **default export** instead. For instance, let's say the only thing we plan to export from `triangle.js` is the `Triangle` constructor function. We can do the following:

triangle.js

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

Remember, this only works if we are exporting only one thing from a file — and we must include the `default` keyword.

When we use default exports, we have to make a small but significant change to our import statements as well:

index.js

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

Note that we no longer use curly braces `{ }` around the thing we are importing. In fact, because we are importing a default, we don't even need to call it `Triangle`. We could do something like this:

index.js

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

It is very important to be able to distinguish between named exports and default exports.

We *can* do the following with a default export: `import Thingy from './../js/shape.js';`. That's because JavaScript knows that only one thing is being imported from the file so we can call it whatever we want. However, it's better to keep the naming consistent to avoid confusion. That means we will generally keep the name exactly the same even though we don't have to.

Make sure you understand the difference between named exports and default exports — as well as the subtle but significant difference between the import syntax for both. These little differences trip up many developers.

While ES6's implementation of `import` and `export` statements is really just syntactic sugar over the traditional way of doing things (using `require` statements), utilizing these features can make your

code cleaner, more organized, and easier to read. You should recognize both ways of importing and exporting code, but we'll focus on using `import` and `export` while at Epicodus.

One other thing to note: some browsers don't understand `require` statements or `import` and `export` statements. Evergreen browsers like Chrome do, but in the real world, many people aren't using Chrome — and may even be using a legacy browser that doesn't support this syntax. Fortunately, webpack will take care of that for us! This is where **concatenating** files comes into the picture. We'll cover that in the next lesson.

Previous (/intermediate-javascript/test-driven-development-and-environments-with-javascript/configuring-webpack-and-using-npm-scripts)
Next (/intermediate-javascript/test-driven-development-and-environments-with-javascript/bundling-javascript)

Lesson 11 of 49
Last updated more than 3 months ago.

disable dark mode

(http://www.epicodus.com)