Organizing Code with Import/Export

 (https://github.com/learn-co-curriculum/react-hooks-import-export)  (https://github.com/learn-co-curriculum/react-hooks-import-export/issues/new)

Learning Goals

- Understand why it's important to split up our code into smaller files.
- Learn how `import` and `export` support our ability to build modular code.
- Understand the different ways to import and export code.

Introduction

In this lesson, we'll discuss the `import` and `export` keywords and how they allow us to share JavaScript code across multiple files.

Modular Code

Modular code is code that is separated into segments (modules), where each file is responsible for a feature or specific functionality.

Developers separate their code into modules for many reasons:

- **Stricter variable scope**: Variables declared in modules are private unless they are explicitly exported, so by using modules, you don't have to worry about polluting the global variable scope.
- **Single-responsibility principle**: Each module is responsible for accomplishing a certain piece of functionality, or adding a specific feature to the application.
- **Easier to navigate**: Modules that are separated and clearly named make code more readable for other developers.
- **Easier to debug**: Bugs have less room to hide in isolated, contained code.
- **Produce clean and DRY code**: Modules can be reused and repurposed throughout applications.

Modularizing React Code

React makes the modularization of code easy by introducing the component structure.

```
function ColoradoStateParks() {
  return <h1>Colorado State Parks!</h1>;
}
```

It's standard practice to give each of these components their own file. It is not uncommon to see a React program file tree that looks something like this:

```
├── README.md
├── public
└── src
     ├── App.js
     ├── ColoradoStateParks.js
     └── index.js
```

With our components separated in their own files, all we have to do is figure out how to access the code defined in one file within a different file. Well, this is easily done in modern JavaScript thanks to the **ES module** (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules) system using the `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) keywords!

Import and Export

On a simplified level, the `import` and `export` keywords let us define variables in one file, and access those variables in other files throughout our project. This becomes increasingly important as we build out larger applications. Let's look at how we can do this.

Fork and clone the repo for this lesson if you'd like to follow along with the examples.

Exporting Variables

Since variables in modules are not visible to other modules by default, we must explicitly state which variables should be made available to the rest of our application. Exporting any variable — whether that variable is an object, string, number, function, or React component — allows us to access that exported variable in other files.

There are two ways to export code in JavaScript: we can use the default export syntax, or the named export syntax.

Default Export Syntax

We can only use `export default` once per file. This syntax lets us export one variable from a file which we can then import in another file.

For example:

```
// src/parks/howManyParks.js

function howManyParks() {
  console.log("42 parks!");
}

export default howManyParks;
```

Once we've exported the variable, we can use the `import` keyword to access that variable in another file:

```
// src/ColoradoStateParks.js
import React from "react";
import howManyParks from "./parks/howManyParks";

function ColoradoStateParks() {
  howManyParks(); // => "42 parks!"

  return <h1>Colorado State Parks!</h1>;
}
```

The `export default` syntax allows us rename the exported variable to whatever we want when importing it:

```
// src/ColoradoStateParks.js
import React from "react";
import aDifferentName from "./parks/howManyParks";

function ColoradoStateParks() {
  aDifferentName(); // => "42 parks!"

  return <h1>Colorado State Parks!</h1>;
}
```

It's generally not advised to rename exports, since it can make it more difficult to debug. Use this technique sparingly (for example, when you are using a library that exports a default variable with the same name as one of your own variables).

Since React components are also just functions, we can export them too! You'll typically have just one React component per file, so it makes sense to use the `export default` syntax with React components, like so:

```
// src/parks/MesaVerde.js
import React from "react";

function MesaVerde() {
  return <h1>Mesa Verde National Park</h1>;
}

export default MesaVerde;
```

Then, we can import the entire component into any other file in our application, using whatever naming convention that we see fit:

```
// src/ColoradoStateParks.js
import React from "react";
import MesaVerde from "./parks/MesaVerde";

function ColoradoStateParks() {
```

```
    return (
      <div>
        <MesaVerde />
      </div>
    );
  }

  export default ColoradoStateParks;
```

You may come across a slightly different way of writing this, with `export default` written directly in front of the name of the function:

```
export default function ColoradoStateParks() {
  // ...
}
```

Our preferred approach is to write the export statements at the bottom of the file for consistency and to make it easier for other developers to identify what is being exported, but the syntax above will also work.

Named Exports

With named exports, we can export multiple variables from a file, like so:

```
// src/parks/RockyMountain.js
const trees = "Aspen and Pine";

function wildlife() {
  console.log("Elk, Bighorn Sheep, Moose");
}

function elevation() {
  console.log("9583 ft");
}
```

```
// named export syntax:
export { trees, wildlife };
```

We can then `import` and use them in another file:

```
// src/ColoradoStateParks.js
import { trees, wildlife } from "./parks/RockyMountain";

console.log(trees);
// => "Aspen and Pine"

wildlife();
// => "Elk, Bighorn Sheep, Moose"
```

We can also write named exports next to the function definition:

```
// src/parks/RockyMountain.js
export const trees = "Aspen and Pine";

export function wildlife() {
  console.log("Elk, Bighorn Sheep, Moose");
}

function elevation() {
  console.log("9583 ft");
}
```

Import

The `import` keyword lets us take variables that we've exported and use them in other files throughout our application. There are many ways to use the `import` keyword, and the method that we use depends on what `export` syntax we used.

In order to import a variable in another file, we write out the **relative path** to the file that we are trying to access. Let's look at some examples.

Importing Multiple Variables

The `import * from` syntax imports all of the variables that have been exported from a given module. This syntax looks like:

```
// src/ColoradoStateParks.js
import * as RMFunctions from "./parks/RockyMountain";

console.log(RMFunctions.trees);
// => "Aspen and Pine"

RMFunctions.wildlife();
// => "Elk, Bighorn Sheep, Moose"

RMFunctions.elevation();
// => Attempted import error
```

In the example above, we're importing all the exported variables from file `RockyMountain.js` as properties on an object called `RMFunctions`. Since `elevation` is not exported, trying to use that function will result in an error.

We are using the **relative path** to navigate from `src/ColoradoStateParks.js` to `src/parks/RockyMountain.js`. Since our file structure looks like this:

```
└── src
    ├── parks
    |   ├── RockyMountain.js
    |   ├── MesaVerde.js
    |   └── howManyParks.js
    ├── ColoradoStateParks.js
    └── index.js
```

To get from `ColoradoStateParks.js` to `RockyMountain.js`, we can stay in the `src` directory, then navigate to `parks`, where we'll find `RockyMountain.js`.

Importing Specific Variables

The `import { variable } from` syntax allows us to access a specific variable by name, and use that variable within our file.

We're able to reference the imported variable by its previously declared name:

```
// src/ColoradoStateParks.js
import { trees, wildlife } from "./parks/RockyMountain";

console.log(trees);
// > "Aspen and Pine"

wildlife();
// > "Elk, Bighorn Sheep, Moose"
```

We can also rename any or all of the variables inside of our `import` statement:

```
// src/ColoradoStateParks.js
import {
  trees as parkTrees,
  wildlife as parkWildlife,
} from "./parks/RockyMountain";

console.log(parkTrees);
// > "Aspen and Pine"

parkWildlife();
// > "Elk, Bighorn Sheep, Moose"
```

Importing Node Modules

```
// src/ColoradoStateParks.js
import React from "react";
import howManyParks from "./parks/howManyParks";
import MesaVerde from "./parks/MesaVerde";
import * as RMFunctions from "./parks/RockyMountain";

export default function ColoradoStateParks() {
  return (
    <div>
      <MesaVerde />
    </div>
  );
}
```

Take a look at the first line of code in this file: `import React from 'react'` . Here, we are referencing the React library's default export. The React library is located inside of the `node_modules` directory, a specific folder in many Node projects that holds packages of third-party code. Any time we are using code from an npm package, we must also import it in whatever file we're using it in.

> **Note**: The ability to import packages from the `node_modules` directory is made possible in projects created with `create-react-app` by a tool called **webpack (https://webpack.js.org/)** . webpack provides a lot of great features for building applications with JavaScript such as making it convenient to import code from other libraries, but configuring webpack from scratch can be quite challenging! Thankfully, `create-react-app` provides this configuration for us under the hood.

Conclusion

The `import` and `export` keywords help keep our code modular, and use variables across different files. In addition to being able to `import` and `export` default functions, we can rename and alias `import` s. We can also reference external packages, like `react` , that installed in our project.

> **Note**: The `import` and `export` keywords are relatively new features of the JavaScript language, having gained full support in major browsers in 2018 and only being added to Node in 2019. For many years, modular code in JavaScript was possible via libraries and other

> module systems, like CommonJS. You'll likely encounter other versions of import/export syntax, like `require` and `module.exports`, in other JavaScript projects and documentation. For our React lessons, though, `import` and `export` are all you need!

Resources

- **ES modules: A cartoon deep-dive (https://hacks.mozilla.org/2018/03/es-modules-a-cartoon-deep-dive/)**
- **MDN Import Documentation (https://developer.mozilla.org/en-US/docs/web/javascript/reference/statements/import)**
- **MDN Export Documentation (https://developer.mozilla.org/en-US/docs/web/javascript/reference/statements/export)**