Organizing Code with Import/Export

New Attempt

Due No Due Date **Points** 1 **Submitting** a website url

FORK (https://github.com/learn-co-curriculum/react-hooks-import-export/fork) (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
- **Adhere to the 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 Hogwarts() {
  return (
    <div className="Hogwarts">
      "Harry. Did you put your name in the Goblet of Fire?"
    </div>
  );
}
```

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
     ├── Hogwarts.js
     └── Houses.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 using `import` and `export` !

Import and Export

On a simplified level, `import` and `export` enable us to use code from one file in other locations across our projects, which 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.

Export

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 a component — or module of code — allows us to call upon that `export` -ed variable in other files, and use the embedded code within other modules. There are two ways to `export` code in JavaScript: we can use the `export default` syntax or we can explicitly name our exports.

Export Default

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

For example:

```
// src/houses/whoseHouse.js

function whoseHouse() {
  console.log("HAGRID'S HOUSE!");
}

export default whoseHouse;
```

This enables us to use `import` to make use of that function elsewhere:

```
// src/Hogwarts.js
import React from "react";
import whoseHouse from "./houses/whoseHouse";

function Hogwarts() {
  whoseHouse(); // => "HAGRID'S HOUSE!"

  return <h1>Welcome to Hogwarts!</h1>;
}
```

`export default` allows us to name the exported code whatever we want when importing it:

```
// src/Hogwarts.js
import React from "react";
import aDifferentName from "./houses/whoseHouse";

function Hogwarts() {
  aDifferentName(); // => "HAGRID'S HOUSE!"

  return <h1>Welcome to Hogwarts!</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/houses/Hufflepuff.js
import React from "react";

function Hufflepuff() {
  return <div>NOBODY CARES ABOUT US</div>;
}
```

```
export default Hufflepuff;
```

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

```
// src/Hogwarts.js
import React from "react";
import Hufflepuff from "./houses/Hufflepuff";

function Hogwarts() {
  return (
    <div>
      <Hufflepuff />
    </div>
  );
}

export default Hogwarts;
```

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 Hogwarts() {
  // ...
}
```

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 within a module, allowing us to call on them explicitly when we `import` .

Named exports allow us to export several specific things at once:

```
// src/houses/Gryffindor.js
const colors = "Scarlet and Gold";

function values() {
  console.log("Courage, Bravery, Nerve and Chivalry");
}

function gryffMascot() {
  console.log("The Lion");
}
```

```
export { colors, gryffMascot };
```

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

```
// src/Hogwarts.js
import { colors, gryffMascot } from "./houses/Gryffindor";

console.log(colors);
// => 'Scarlet and Gold'

gryffMascot();
// => 'The Lion'
```

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

```
// src/houses/Gryffindor.js
export const colors = "Scarlet and Gold";

function values() {
  console.log("Courage, Bravery, Nerve and Chivalry");
}

export function gryffMascot() {
  console.log("The Lion");
}
```

Import

The `import` keyword is what enables us to take modules that we've exported and use them in other files throughout our applications. There are many ways to `import` with React, and the method that we use depends on what type of code we are trying to access and how we exported it.

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

import * from

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

```
// src/Hogwarts.js
import * as GryffFunctions from "./houses/Gryffindor";

console.log(GryffFunctions.colors);
// > 'Scarlet and Gold'
```

```
GryffFunctions.gryffMascot();
// => 'The Lion'

GryffFunctions.values();
// => Attempted import error
```

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

We are using the **relative path** to navigate from `src/Hogwarts.js` to `src/houses/Gryffindor.js` . Since our file structure looks like this:

```
└── src
     ├── houses
     |    ├── Gryffindor.js
     |    ├── Hufflepuff.js
     |    └── whoseHouse.js
     ├── Hogwarts.js
     └── index.js
```

To get from `Hogwarts.js` to `Gryffindor.js` , we can stay in the `src` directory, then navigate to `houses` , where we'll find `Gryffindor.js` .

import { variable } from

`import { variable } from` allows us to grab a specific variable/function by name, and use that variable/function within the body of a new module.

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

```
// src/Hogwarts.js
import { colors, gryffMascot } from "./houses/Gryffindor";

console.log(colors);
// > 'Scarlet and Gold'

gryffMascot();
// > 'The Lion'
```

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

```
// src/Hogwarts.js
import { colors as houseColors, gryffMascot as mascot } from "./houses/Gryffindor"

console.log(houseColors);
// > 'Scarlet and Gold'
```

```
mascot();
// > 'The Lion'
```

Importing Node Modules

```
// src/Hogwarts.js
import React from "react";
import whoseHouse from "./houses/whoseHouse";
import Hufflepuff from "./houses/Hufflepuff";
import * as GryffFunctions from "./houses/Gryffindor";

export default function Hogwarts() {
  return (
    <div>
      <HooflePoof />
    </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.

Conclusion

`import` and `export` enable us to keep code modular, and use it 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 npm packages that are in our project.

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)