All about modules

Two different standards

In the browser JavaScript ecosystem, JavaScript modules depends on `import` and `export` statements to load and export ES modules.

In addition, ES module is the official standard format to package JavaScript code for reuse.

On the other hand, Node.js, supports the CommonJS module format by default. CommonJS module load using `require()` and the variables and functions export from a CommonJS module with `module.exports`

Why the two different standards? Good question. CommonJS module is built into Node.js before ES module were introduced.

Node.js support ES module

There are two ways you can enable ES modules in Node.js.

First way

You can simply change the file extension from `.js` to `.mjs` to use ES module syntax to load and export modules:

```
// util.mjd
export function add(a, b) {
return a + b;
}
```

```
// app.mjs
import {add} from './util.mjs'


console.log(add(1, 2)); // 3
```

Second way (better way)

You can add a `"type": "module"` field inside the nearest `package.json` file. By including that field, Node.js will treat all files inside the package as ES modules instead, so you wouldn't have to change to `.mjs` extensions.

How frameworks deal with this

You will be using `import/export` for frameworks like React and Vue.js, the framework themselves will use a transpiler to compile the `import/export` syntax down to `require` anyway.

Module

What is a module? A module is just a file, one script is one module. Simple.

Modules can load each other using `export` and `import` directives to give and exchange functionality between different files.

- `export`: This keyword labels variables and functions that should be accessible from outside
- `import`: This keyword allows you to import functionality that are exported by other modules.

Example

```
// sayHi.js
export function sayHi(user) {
console.log("Hello " + user)
}
```

Another file can import it and use it

```
// main.js
import {sayHi} from './sayHi.js'

console.log(sayHi); // function
sayHi("Ricky"); // Hello Ricky
```

Base modules

Any `import` statement must get either a relative or absolute URL. Modules without any path are called bare modules:

```
import {sayHi} from 'sayaHi'
```

They are not allowed in browser, but Node.js or bundle tools allow such bare modules

Build tools

Using a build tool like Webpack will allow you to use bare modules. It also does code optimization and remove unreachable code.

Export/import

Export and import directives have many syntax variants and we will go over them.

Export before declaration

Here is how you export along with the declaration of variables and functions:

```
export let months = [1, 2, 3, 4]; // exporting an array

export const MODULE_CONST = 69; // exporting a const

// exporting a class
export class User {
constructor(name) {
    this.name = name;
}
}

// exporting a function
export function foo() {
console.log("fooing around");
}
```

Export after declaration

Here is how you export if you already have the declaration of variables and functions already:

```
let x = 69;

function sayHi() {
console.log("Hi");
}

function sayBye() {
console.log("Bye");
}

export {x, sayHi, sayBye}; // you pass in a list of exported variables or functions
```

Export as

When you export a variable or function you can also choose a different name to export under, so that the modules that will be importing will use the name that you choose

```
// say.js
export { sayHi as hi, sayBye as bye};
```

```
// main.js
import {hi, bye} from './say.js'

hi();
bye();
```

Export default

Typically there is two types of modules

1. Module that contain a library like bunch of functions
2. Or modules that declare a single entity which exports say only a class for others to use

The second approach is mostly done. And to do it there is the syntax `export default` to export a default export, and there is only one default export per file, one thing that you can defaulty export per module.

```
// user.js
export default class User {
  constructor(name) {
    this.name = name;
  }
}

// or equivalent
class User {
  constructor(name) {
      this.name = name;
    }
}

export {User as default};
```

```
import User from './user.js' // not {User} just User
import {default as User} from './user.js' // this is the second way

new User("ricky");
```

When you are importing the default export you do not need any curly braces and it looks nicer.

On the other hand, named exports needs curly braces, and default export do not need them

> When you are using default exports, you always choose the name when you are importing, it doesn't matter what the name is. However, naming it different things might end up confusing some team members, so the best practice is to name the default export the same as the file names.

```
import User from './user.js'
import LogicForm from './logicForm.js'
import func from '/path/to/func.js'
```

Import

Usually you can put a list of what you want to import in curly braces

```
import { sayHi, sayBye, x} from './say.js'
```

Import *

But if there is a lot to import you can import everything as an object using `import * as <obj>`

```
import * as say from './say.js'

say.sayHi();
say.sayBye();
say.default; // to  access the default export if you export everything *
```

Import <> as

You can also pick an alias for the functions or variables that you have imported using `as`

```
import {sayHi as hi, sayBye as bye} from './say.js'

hi();
bye();
```

Import default export along with name export

The `default` keyword is used to reference to the default export

```
import {default as User, sayHi} from './user.js'
```

Re-exporting

Re-exporting syntax `export ... from` allows you to import things and immediately export them like so:

```
export {sayHi} from './say.js' // re-export sayHi


export {default as User} from './user.js' // re-export the default export under User name
```

Why would you do this? Well imagine you are writing a package: folder with lots of modules and some under a different folder because they are just helper functions. So your file structure could be like so:

```
auth/
    index.js
    user.js
    helpers.js
    tests/
        login.js
    providers/
        github.js
        facebook.js
        ...
```

You would like to expose the package functionality via just a single entry point, i.e. if someone wants to use your package they can just import only from `auth/index.js`.

```
import {login, logout} from 'auth/index.js'
```

Instead of doing it from the exact file that it was exported. We can just let the main file to export all the functionality that we want to provide in the package.

Other programmer who want to use the package shouldn't need to look into the internal structure, search for files inside the package folder for the exact one to import. They can just look at one main file to import from, while keeping the rest hidden.

This is where re-exporting can be used to do this

```
// auth/index.js
// import login/logout and export them
import {login, logout} from './helpers.js'
export {login, logout}

// import default as User and then export it
import User from './user.js'
export {User}
```

Now user can just do `import {login} from 'auth/index.js'`

The syntax `export ... from ...` is just a shorter notation for such import-export:

```
import {login, logout} from './helpers.js'
export {login, logout}

import User from './user.js'
export {User}

// Equivalent to
export {login, logout} from './helpers'.js'
export {default as User} from './user.js' // This is default re-exporting
```

If you are re-exporting a default export using the shorter notation, you must do it via `export {default} from './file.js'` and if you are planning to export other named exports from the same file you would have to do `export * from './file.js'`.

Basically, for default exports you would have to handle it separately, you cannot just `export *`, it will only handle named exports and not default exports.

Revision #2
Created 2022-12-29 23:05:45 UTC by Tamarine
Updated 2022-12-30 14:59:45 UTC by Tamarine