Implementing Modules in Node

Article on modular programs and implementing modules in the Node runtime environment.

What are Modules?

Modules are reusable pieces of code in a file that can be exported and then imported for use in another file. A modular program is one whose components can be separated, used individually, and recombined to create a complex system.

Consider the diagram below of an imaginary program written in a file my\_app.js:

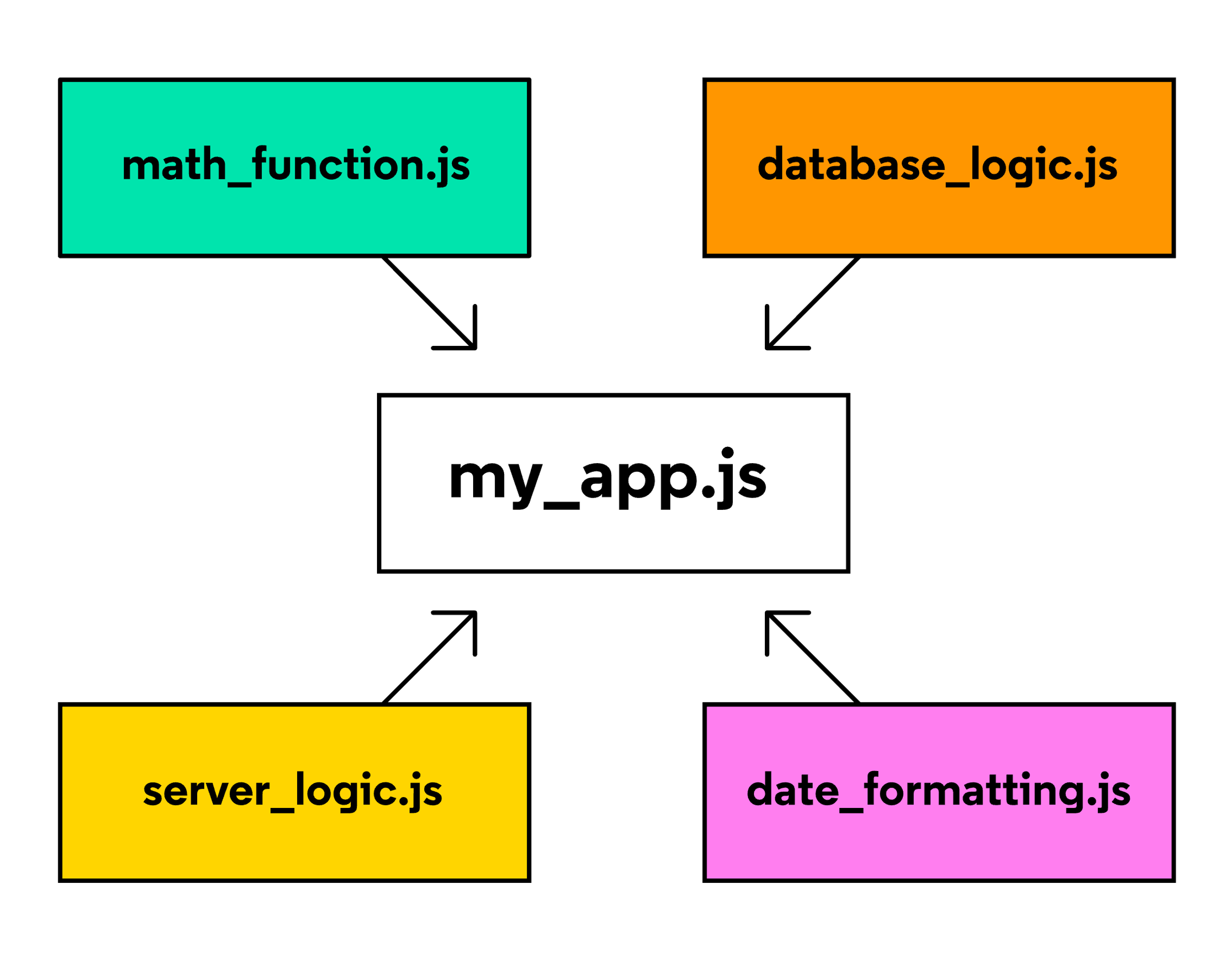


diagram of a modular program. separate modules are combined to create a complex program

Note: The words “module” and “file” are often used interchangably

Instead of having the entire program written within my\_app.js, its components are broken up into separate modules that each handle a particular task. For example, the database\_logic.js module may contain code for storing and retrieving data from a database. Meanwhile, the date\_formatting.js module may contain functions designed to easily convert date values from one format to another (a common headache among programmers!).

This modular strategy is sometimes called separation of concerns and is useful for several reasons. What do you think those reasons might be?

Write down a few of your ideas before revealing the reasons below:

Reusable code,

Logic is separated,

Clean, clear and concise code,

Easy to debug,

Code dependencies become less if all code in one file.

Implementing modules in your program requires a small bit of management. In the remainder of this article, we will be covering:

How to use the Node.js module.exports object to export code from a file - meaning its functions and/or data can be used by other files/modules.

How to use the Node.js require() function to import functions and/or data from another module.

# Implementations of Modules in JavaScript: Node.js vs ES6

Before we dive in, it should be noted that there are multiple ways of implementing modules depending on the runtime environment in which your code is executed. In JavaScript, there are two runtime environments and each has a preferred module implementation:

* The Node runtime environment and the module.exports and require() syntax.
* The browser’s runtime environment and the ES6 import/export syntax.

This article will focus on using the module.exports and require() syntax in the Node runtime environment. For more information, you can read the two articles linked below

## Implementing modules using ES6 Syntax: Introduction to JavaScript Runtime Environments

Implementing Modules in Node

Every JavaScript file that runs in a Node environment is treated as a distinct module. The functions and data defined within each module can be used by any other module, as long as those resources are properly exported and imported.

Suppose you wanted to write a simple program that would display the freezing point and boiling point of water in Fahrenheit. However, you only know the values in Celsius to be 0 (freezing) and 100 (boiling). Luckily you happen to know how to convert Celsius to Fahrenheit!

Such a program might look like this:

/\* water-limits.js \*/

function celsiusToFahrenheit(celsius) {

return celsius \* (9/5) + 32;

}

const freezingPointC = 0;

const boilingPointC = 100;

const freezingPointF = celsiusToFahrenheit(freezingPointC);

const boilingPointF = celsiusToFahrenheit(boilingPointC);

console.log(`The freezing point of water in Fahrenheit is ${freezingPointF}`);

console.log(`The boiling point of water in Fahrenheit is ${boilingPointF}`);

This water-limits.js program is simple but let’s break it down into its parts:

At the top of the file, the function celsiusToFahrenheit() is declared. When given a value in Celsius, it will return the value converted to Fahrenheit. Both input and output will be a number.

Below, freezingPointC and boilingPointC are assigned the known values 0 and 100, respectively.

Using these values and the function celsiusToFahrenheit(), the additional values freezingPointF and boilingPointF are calculated.

Lastly, these values are printed to the console.

Executing this file using Node would look something like this:

$ node water-limits.js

The freezing point of water in Fahrenheit is 32

The boiling point of water in Fahrenheit is 212

Now, you decide to write a second program. In this program, the user can input any temperature value in Celsius and the program responds by printing the input temperature converted to Fahrenheit.

For example, you might want to be able to run such a program and see a response like so:

$ node celsius-to-fahrenheit.js 100

100 degrees Celsius = 212 degrees Fahrenheit

The JavaScript below would do just that:

/\* celsius-to-fahrenheit.js \*/

function celsiusToFahrenheit(celsius) {water

return celsius \* (9/5) + 32;

}

const celsiusInput = process.argv[2]; // Get the 3rd input from the argument list

const fahrenheitValue = celsiusToFahrenheit(celsiusInput);

console.log(`${celsiusInput} degrees Celsius = ${fahrenheitValue} degrees Fahrenheit`);

Now, let’s break down the celsius-to-fahrenheit.js program:

At the top of the file, the function celsiusToFahrenheit() is declared. When given a value in Celsius, it will return the value converted to Fahrenheit. Both input and output will be a number.

On the next line of code, celsiusInput is assigned process.argv[2]. When a program is executed in the Node environment, process.argv is an array holding the arguments provided. In this case, it looks like ['node', 'celsius-to-fahrenheit.js', '100']. So, process.argv[2] returns 100.

Using this value and the function celsiusToFahrenheit(), the additional value fahrenheitValue is calculated.

Lastly, a message is printed to the console displaying this data.

Notice anything similar between the two programs, water-limits.js and celsius-to-fahrenheit.js? Both programs implement the function celsiusToFahrenheit()! Not only did we write this function twice, but if we ever need to make changes to the function we’ll also have to make those changes in two places.

Creating a module that exports a celsiusToFahrenheit() function that can be used by both of these programs would solve this repetitive code problem.

module.exports

To create a module, you simply have to create a new file where the functions can be declared. Then, to make these functions available to other files, add them as properties to the built-in module.exports object:

/\* converters.js \*/

function celsiusToFahrenheit(celsius) {

return celsius \* (9/5) + 32;

}

module.exports.celsiusToFahrenheit = celsiusToFahrenheit;

module.exports.fahrenheitToCelsius = function(fahrenheit) {

return (fahrenheit - 32) \* (5/9);

};

The code snippet above demonstrates two ways of exporting functions from a module. Let’s break it down:

At the top of the new file, converters.js, the function celsiusToFahrenheit() is declared.

On the next line of code, the first approach for exporting a function from a module is shown. In this case, the already-defined function celsiusToFahrenheit() is assigned to module.exports.celsiusToFahrenheit.

Below, an alternative approach for exporting a function from a module is shown. In this second case, a new function expression is declared and assigned to module.exports.fahrenheitToCelsius. This new method is designed to convert Fahrenheit values back to Celsius.

Both approaches successfully store a function within the module.exports object.

module.exports is an object that is built-in to the Node.js runtime environment. The exported values are stored in it as properties and values. Other files can now import this object, and make use of these two functions, with another feature that is built-in to the Node.js runtime environment: the require() function.

require()

The require() function accepts a string as an argument. That string provides the file path to the module you would like to import.

Let’s update water-limits.js such that it uses require() to import the .celsiusToFahrenheit() method from the module.exports object within converters.js:

/\* water-limits.js \*/

const converters = require('./converters.js');

const freezingPointC = 0;

const boilingPointC = 100;

const freezingPointF = converters.celsiusToFahrenheit(freezingPointC);

const boilingPointF = converters.celsiusToFahrenheit(boilingPointC);

console.log(`The freezing point of water in Fahrenheit is ${freezingPointF}`);

console.log(`The boiling point of water in Fahrenheit is ${boilingPointF}`);

In this case, ./ is a relative path indicating that converters.js is stored in the same folder as water-limits.js. When you use require(), the entire module.exports object is returned and stored in the variable converters. This means that both the .celsiusToFahrenheit() and .fahrenheitToCelsius() methods can be used in this program!

Using Object Destructuring to be more Selective With require()

In many cases, modules will export a large number of functions but only one or two of them are needed. You can use object destructuring to extract only the needed functions.

Let’s update celsius-to-fahrenheit.js and only extract the .celsiusToFahrenheit() method, leaving .fahrenheitToCelsius() behind:

/\* celsius-to-fahrenheit.js \*/

const { celsiusToFahrenheit } = require('./converters.js');

const celsiusInput = process.argv[2];

const fahrenheitValue = celsiusToFahrenheit(input);

console.log(`${celsiusInput} degrees Celsius = ${fahrenheitValue} degrees Fahrenheit`);

With this approach, the remainder of the program works the same way but the program avoids importing a function it does not need.

Implementing Modules using ES6 Syntax

Article on implementing modules in a browser’s runtime environment using ES6 modules syntax.

What are Modules?

Modules are reusable pieces of code in a file that can be exported and then imported for use in another file. A modular program is one whose components can be separated, used individually, and recombined to create a complex system.

Consider the diagram below of an imaginary program written in a file my\_app.js:

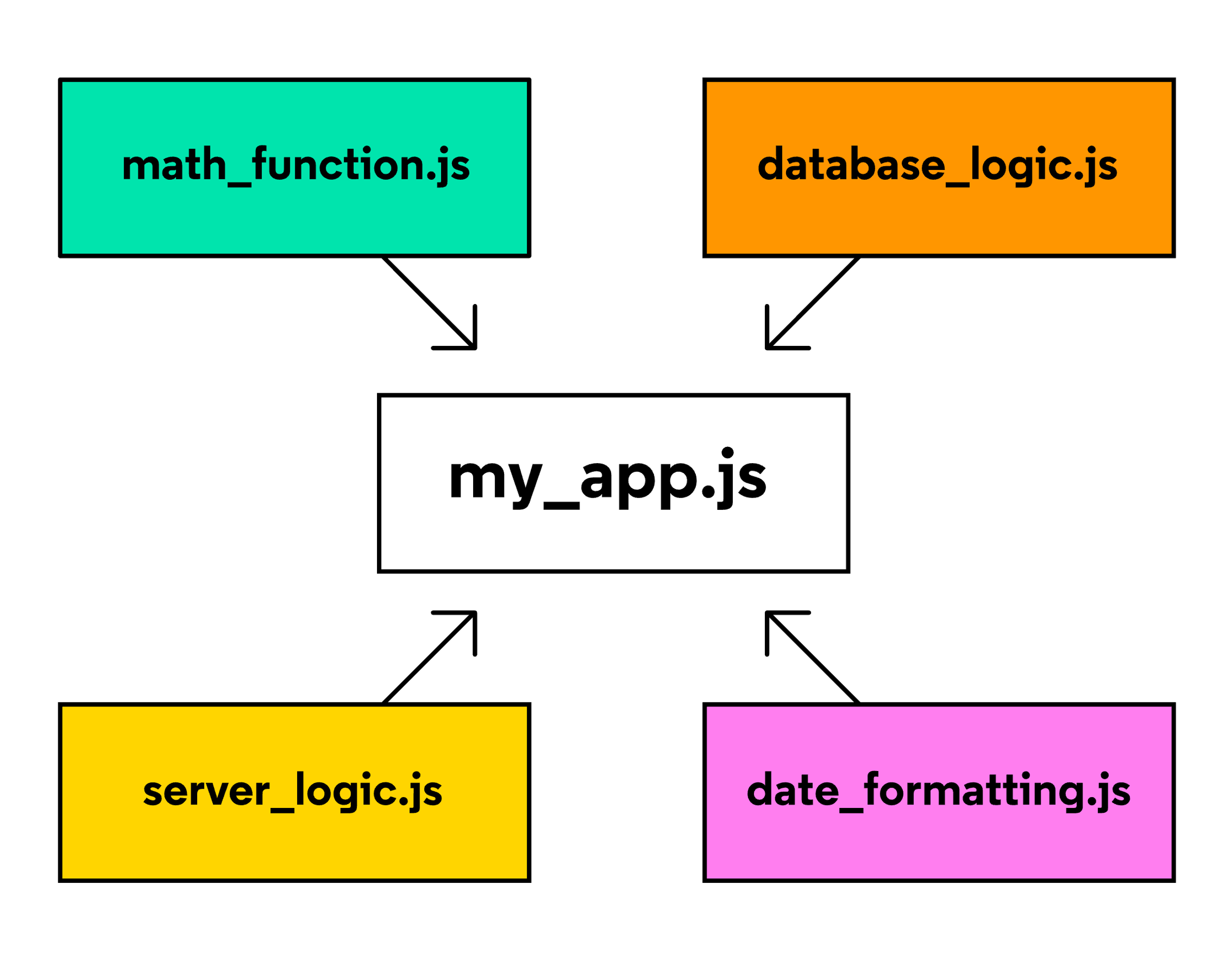


diagram of a modular program. separate modules are combined to create a complex program

Note: The words “module” and “file” are often used interchangably

Instead of having the entire program written within my\_app.js, its components are broken up into separate modules that each handle a particular task. For example, the database\_logic.js module may contain code for storing and retrieving data from a database. Meanwhile, the date\_formatting.js module may contain functions designed to easily convert date values from one format to another (a common headache among programmers!).

This modular strategy is sometimes called separation of concerns and is useful for a number of reasons. What do you think those reasons might be?

Write down a few of your ideas before revealing the reasons below:

Implementing modules in your program requires a small bit of management. In the remainder of this article, we will be covering:

How to use the ES6 export statement to export code from a file - meaning its functions and/or data can be used by other files/modules.

How to use the ES6 import statement to import functions and/or data from another module.

Implementations of Modules in JavaScript: Node.js vs ES6

Before we dive in, it should be noted that there are multiple ways of implementing modules depending on the runtime environment in which your code is executed. In JavaScript, there are two runtime environments and each has a preferred module implementation:

The Node runtime environment and the module.exports and require() syntax.

The browser’s runtime environment and the ES6 import/export syntax.

This article will focus on using the ES6 import/export syntax in a browser’s runtime environment. For more information, you can read the articles linked below.

# Introduction to Runtime Environments

A Brief History of JavaScript Modules in the Browser

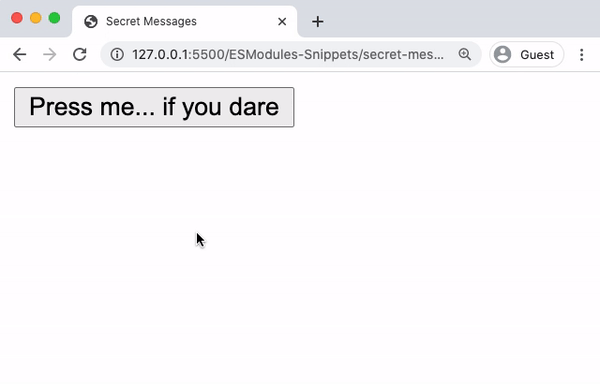
In the early days of web development, JavaScript usage was fairly minimal. A script here to add some interactivity to a page and a script there to automate away some simple task. Nowadays, however, JavaScript dominates the web and scripts have ballooned into large and cumbersome behemoths. According to some studies, the average size of a website, in terms of kilobytes of JavaScript data transferred, has grown over 5 times from 2010 to 2020!

These stats aren’t meant to paint a dreary future of web development. Web applications drive much of modern society and are far more capable than could have been imagined when the World Wide Web was created in 1989. Instead, it is to make clear the need for modularity as the capabilities, and size, of these scripts grow.

Though libraries for implementing modules have existed for some time, syntax for natively implementing modules was only introduced in 2015 with the release of ECMAScript 6 (ES6). Since then, it has been adopted by most modern browsers and is the de facto approach for implementing modular applications in the browser.

Implementing Modules in the Browser

Let’s take a look at implementing modules in the browser through an example. Suppose you wanted to build a simple web application with some hidden text that is revealed when a button is pressed.



A demo of a simple website. There is a button that says "Press me... if you dare". Clicking on the button reveals a hidden message that says "Modules are fancy!"

To create this website, you could create two files, secret-messages.html and secret-messages.js, and store them together in a folder called secret-messages:

secret-messages/

|-- secret-messages.html

|-- secret-messages.js

Let’s take a look at the HTML file first:

<!-- secret-messages.html -->

<html>

<head>

<title>Secret Messages</title>

</head>

<body>

<button id="secret-button"> Press me... if you dare </button>

<p id="secret-p" style="display: none"> Modules are fancy! </p>

<script src="./secret-messages.js"> </script>

</body>

</html>

The secret-messages.html page renders a button element and a hidden paragraph element.

It then loads the script secret-messages.js using the file path "./secret-messages.js". The ./ before the file name is how you indicate that the file being referenced (secret-messages.js) is in the same folder as the file referencing it (secret-messages.html).

Speaking of which, let’s take a look at the JavaScript file:

/\* secret-messages.js \*/

const buttonElement = document.getElementById('secret-button');

const pElement = document.getElementById('secret-p');

const toggleHiddenElement = (domElement) => {

if (domElement.style.display === 'none') {

domElement.style.display = 'block';

} else {

domElement.style.display = 'none';

}

}

buttonElement.addEventListener('click', () => {

toggleHiddenElement(pElement);

});

In secret-messages.js, DOM objects are created to reference the button element and paragraph element using the DOM API. These objects are stored in buttonElement and pElement, respectively.

The function toggleHiddenElement() is declared. It can accept either of these elements as an input called domElement and will either show or hide that element depending on its current style.display value.

An event listener is added to buttonElement to listen for 'click' events and respond by calling toggleHiddenElement() with pElement as the argument.

Now, suppose you wanted to create a second webpage with similar features. There is still a button, but this time clicking it reveals an image. Using similar logic as the program above, this can be achieved with the following file structure:

secret-image/

|-- secret-image.html

|-- secret-image.js

The HTML might look like this:

<!-- secret-image.html -->

<html>

<head>

<title>Secret Image</title>

</head>

<body>

<button id="secret-button"> Want to see something cool? </button>

<img id="secret-img" src="imageURL.jpg" style="display: none">

<script src="./secret-image.js"> </script>

</body>

</html>

… and the JavaScript might look like this:

/\* secret-image.js \*/

const buttonElement = document.getElementById('secret-button');

const imgElement = document.getElementById('secret-img');

const toggleHiddenElement = (domElement) => {

if (domElement.style.display === 'none') {

domElement.style.display = 'block';

} else {

domElement.style.display = 'none';

}

}

buttonElement.addEventListener('click', () => {

toggleHiddenElement(imgElement);

});

Given that much of the code in these two programs is similar, creating this second website was fairly straightforward. In particular, notice that the toggleHiddenElement() function is copied line for line from secret-messages.js.

Having two identical, but separate, copies of a function can lead to maintenance issues in the future. For example, any bugs that may exist within the function would need to be fixed in two places rather than one.

Instead, creating a single copy of toggleHiddenElement() within a module that exports it would allow these two websites to import and use the exact same function. With this approach, updates to the function only need to occur within the module that defines them, and all programs that import this function will receive the same update. Furthermore, additional functions could be exported by the module and used by both programs, further reducing repetition.

ES6 Named Export Syntax

A module must be entirely contained within a file. So, let’s first consider where a new module may be placed within the file system. Since it needs to be used by both of these projects, you may want to put it in a mutually accessible location. The entire file structure containing both projects and this new module, let’s call it dom-functions.js, could look like this:

secret-image/

|-- secret-image.html

|-- secret-image.js

secret-messages/

|-- secret-messages.html

|-- secret-messages.js

modules/

|-- dom-functions.js <-- new module file

Inside dom-functions.js, the functions you wish to reuse can be exported using the named export syntax below:

export { resourceToExportA, resourceToExportB, ...}

Using this syntax, the name of each exported resource is listed between curly braces and separated by commas. Below, you can see how this is implemented in the new module file dom-functions.js:

/\* dom-functions.js \*/

const toggleHiddenElement = (domElement) => {

if (domElement.style.display === 'none') {

domElement.style.display = 'block';

} else {

domElement.style.display = 'none';

}

}

const changeToFunkyColor = (domElement) => {

const r = Math.random() \* 255;

const g = Math.random() \* 255;

const b = Math.random() \* 255;

domElement.style.background = `rgb(${r}, ${g}, ${b})`;

}

export { toggleHiddenElement, changeToFunkyColor };

Let’s briefly break down how this module works:

The function toggleHiddenElement() is declared. It accepts a domElement as an input and either shows or hides that element depending on its current display style value.

A new function changeToFunkyColor() is declared. It accepts a domElement as an input and then sets its background color to a random rgb() color value.

The two functions are exported using the ES6 export statement.

These exported functions are now available to be imported and used by other files!

If you want to try this out on your own computer, you will need to spin up a local server or else you will run into CORS issues. Check out this article on creating a local server with VSCode and the Live Server extension.

In addition to the syntax above, in which all named exports are listed together, individual values may be exported as named exports by simply placing the export keyword in front of the variable’s declaration. Here is the same example using this syntax:

/\* dom-functions.js \*/

export const toggleHiddenElement = (domElement) => {

// logic omitted...

}

export const changeToFunkyColor = (domElement) => {

// logic omitted...

}

ES6 Import Syntax

The ES6 syntax for importing named resources from modules is similar to the export syntax:

import { exportedResourceA, exportedResourceB } from '/path/to/module.js';

Let’s update the secret-messages program such that it now imports functionality from dom-functions.js. Doing so requires two important steps. First, update secret-messages.js:

/\* secret-messages.js \*/

import { toggleHiddenElement, changeToFunkyColor } from '../modules/dom-functions.js';

const buttonElement = document.getElementById('secret-button');

const pElement = document.getElementById('secret-p');

buttonElement.addEventListener('click', () => {

toggleHiddenElement(pElement);

changeToFunkyColor(buttonElement);

});

Let’s break down these changes:

In secret-messages.js, the functions toggleHiddenElement() and changeToFunkyColor() are imported from the module ../modules/dom-functions.js. The ../ indicates that the modules/ folder is in the same folder as the parent folder, secret-messages/.

When the button is clicked, the now imported toggleHiddenElement() function is called with pElement as an argument.

In addition, changeToFunkyColor() is called with buttonElement as an argument, changing its background color to a random one!

Now, you must also update secret-messages.html:

<!-- secret-messages.html -->

<html>

<head>

<title>Secret Messages</title>

</head>

<body>

<button id="secret-button"> Press me... if you dare </button>

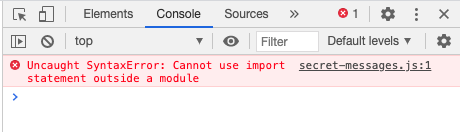
<p id="secret-p" style="display: none"> Modules are fancy! </p>

<script type="module" src="./secret-messages.js"> </script>

</body>

</html>

The change here is subtle, can you spot it? In secret-messages.html, the only thing that changes is the addition of the attribute type='module' to the <script> element. Failure to do so can cause some browsers to throw an error. For example, in Chrome you might see this error:



Uncaught SyntaxError: Cannot use import statement outside a module

And those are the basics of exporting and importing using the ES6 export and import syntax! If you have been following along with these code examples, see if you can update the secret-image project to use the exported functions from the module dom-functions.js before continuing on to the challenges below.

# Renaming Imports to Avoid Naming Collisions

Inevitably, you will run into a situation where the resources you wish to import share a name with some other value that already exists in your program (or from another imported module).

For example, suppose you had access to two modules, greeterEspanol.js and greeterFrancais.js. Each exports a function called greet():

/\* inside greeterEspanol.js \*/

const greet = () => {

console.log('hola');

}

export { greet };

/\* inside greeterFrancais.js \*/

const greet = () => {

console.log('bonjour');

}

export { greet };

Now, let’s say you wanted to use each of these functions in a program, called main.js, that simulates a conversation between a spanish-speaker and a french-speaker.

import { greet } from 'greeterEspanol.js';

import { greet } from 'greeterFrancais.js';

The code above will throw an error on line 2 due to the fact that the identifier greet has already been defined on line 1. Thankfully, ES6 includes syntax for renaming imported resources by adding in the keyword as. It looks like this:

import { exportedResource as newlyNamedResource } from '/path/to/module'

Let’s see how this could work within main.js

/\* main.js \*/

import { greet as greetEspanol } from 'greeterEspanol.js';

import { greet as greetFrancais } from 'greeterFrancais.js';

greetEspanol(); // Prints: hola

greetFrancais(); // Prints: bonjour

Now, both of the imported functions can be called within main.js using their new identifiers, greetEspanol() and greetFrancais().

ES6 Modules Challenge #4

Fill in the Code

Consider the following functions being exported from the modules square-area.js and circle-area.js:

/\* From square-area.js... \*/

export function area(side) {

return side \* side;

}

/\* From circle-area.js... \*/

export function area(radius) {

return Math.PI \* r \* r;

}

The file below, area-calculator.js, is programmed to print the area of a square and a circle to the console using these functions. However, because the exported values share the same name, they need to be renamed when they are imported! Fill in the code blanks below with the correct syntax for renaming these functions.

/\* area-calculator.js \*/

import { } from 'square-area.js';

import { } from 'circle-area.js';

console.log('The area of a square with sides of length 5 is ' + squareArea(5));

console.log('The area of a circle with radius of length 5 is ' + circleArea(5));

area as squareArea

circleArea

area as circleArea

squareArea

Click or drag and drop to fill in the blank

# Default Exports and Imports

So far, the examples shown have used the named export syntax, in which a module’s resources are exported individually by name. Every module also has the option to export a single value to represent the entire module called the default export. Often, though not always, the default export value is an object containing the entire set of functions and/or data values of a module.

The syntax for exporting a default object looks like this:

const resources = {

valueA,

valueB

}

export { resources as default };

With this syntax, the object containing the module’s resources is first declared and then is exported on the next line. At first glance, it looks like the resources object is being exported as a named export. However, the clause as default renames the exported object to default, a reserved identifier that can only be given to a single exported value.

You may also see this shorthand syntax where the value follows export default is the default value to be exported:

const resources = {

valueA,

valueB

}

export default resources;

# The syntax for importing default exports looks like this:

import importedResources from 'module.js';

Notice that the curly braces are gone from the import statement. This syntax is actually shorthand for

import { default as importedResources } from 'module.js

and the imported default value may be given any name the programmer chooses.

It should be noted that if the default export is an object, the values inside cannot be extracted until after the object is imported, like so:

// This will work...

import resources from 'module.js'

const { valueA, valueB } = resources;

// This will not work...

import { valueA, valueB } from 'module.js'

Let’s return to the prior example. The dom-functions.js module from above could be rewritten to use default exports like so:

/\* dom-functions.js \*/

const toggleHiddenElement = (domElement) => {

if (domElement.style.display === 'none') {

domElement.style.display = 'block';

} else {

domElement.style.display = 'none';

}

}

const changeToFunkyColor = (domElement) => {

const r = Math.random() \* 255;

const g = Math.random() \* 255;

const b = Math.random() \* 255;

domElement.style.background = `rgb(${r}, ${g}, ${b})`;

}

const resources = {

toggleHiddenElement,

changeToFunkyColor

}

export default resources;

This default exports object can now be used within secret-messages.js like so:

import domFunctions from '../modules/dom-functions.js';

const { toggleHiddenElement, changeToFunkyColor } = domFunctions;

const buttonElement = document.getElementById('secret-button');

const pElement = document.getElementById('secret-p');

buttonElement.addEventListener('click', () => {

toggleHiddenElement(pElement);

changeToFunkyColor(buttonElement);

});

Notice how toggleHiddenElement() and changeToFunkyColor() are now methods of the imported object called domFunctions and are extracted using ES6 destructuring syntax! It should be noted that the identifier domFunctions can be chosen as the programmer sees fit. If you recall, the syntax used in the snippet above is shorthand for:

import { default as domFunctions } from '../modules/dom-functions.js';

Continue on to the challenges below before the final review at the end of the article.

ES6 Modules Challenge #5

Code Challenge

In this module you will find two functions which have been declared for you, changeText() and changeToFunkyColor(). The website being rendered wants to make use of these functions but currently, they aren’t being exported.

Using the default export syntax, export an object representing the module and containing the changeText() and changeToFunkyColor() functions.

Note: Upon completing this challenge, the text will change to "You did it" and will be set to change color every 0.2 seconds.

12345678910111213141516

function changeText(domElement, newText) {

domElement.innerHTML = newText;

}

function changeToFunkyColor(domElement) {

const r = Math.random() \* 255;

const g = Math.random() \* 255;

const b = Math.random() \* 255;

domElement.style.color = `rgb(${r}, ${g}

Run

ES6 Modules Challenge #6

Multiple Choice Question

Consider the module below which exports a few values using the default export syntax:

/\* article-data.js \*/

const resources = {

articleTitle: "Implementing Modules using ES6 Syntax",

numberOfChallenges: 6,

minutesToComplete: 45

}

export default resources;

Which of the following is NOT a valid statement for importing this data?

import resources from './article-data.js'

import articleData from './article-data.js'

import { articleTitle, numberOfChallenges, minutesToComplete } from './article-data.js'

import { default as articleData } from './article-data.js'

Review

In this article, you have learned the following:

The benefits of implementing modular programs.

How to use the ES6 export statement to export code from a file - meaning its functions and/or data can be used by other files/modules.

How to use the ES6 import statement to import functions and/or data from another module.

How to rename imported resources using the as keyword.

How to export and import a default value.

Though this article covers the basics of using ES6 syntax to import and export modules, MDN has an excellent article that provides an in-depth look at some additional features that ES6 has to offer.