Trabajando con imports

Cuando trabajamos con javascript, vamos a necesitar organizar nuestro código para separar funcionalidades en distintos ficheros, y acceder a ella desde otro fichero diferente.

Importando ficheros desde html

Cada fichero que deseamos importar lo añadimos con una etiqueta *script*:

```
<script src="..."></script>
```

Así, como en el siguiente ejemplo, podemos tener nuestros *scripts* de javascript importados desde el propio html:

index.html

```
<!DOCTYPE html>
<html lang="es">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0"
/>
    <meta http-equiv="X-UA-Compatible" content="ie=edge" />
    <title>Trabajando con imports</title>
    <script src="./utils.js"></script>
    <script src="./index.js"></script>
  </head>
  <body>
    <div>Bootcamp Javascript Lemoncode - Trabajando con imports</div>
  </body>
</html>
```

En el anterior html, importamos los script de *utils.js* y *index.js* al cargar el documento html. Como ejemplo, tendremos el siguiente contenido en cada uno:

utils.js

```
function hello() {
  return "¡Hola! :)";
}
```

index.js

```
document.write(hello());
```

Son dos scripts sencillos. En *utils.js* tenemos una función *hello* que devuelve un texto de saludo "¡Hola! 😃". En nuestro fichero *index.js* llamaremos a esa función definida en el fichero anterior de *hello.js* para imprimir su resultado en el documento.

Problemas con esta forma de trabajar:

- Ensucia nuestro html.
- Hay que tener en cuenta el orden en el que importamos los scripts.
- Alta posibilidad de tener colisiones globales en nuestro documento con funcionalidad con el mismo nombre.

> ¿Qué ocurre si cambiamos el orden de los ficheros javascript en el html?

Vamos a probar a cambiar el orden de importación de los scripts en el fichero *index.html*:

```
<!DOCTYPE html>
<html lang="es">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0"
/>
    <meta http-equiv="X-UA-Compatible" content="ie=edge" />
    <title>Trabajando con imports</title>
    <script src="./utils.js"></script>
    <script src="./index.js"></script>
  </head>
  <body>
    <div>Bootcamp Javascript Lemoncode - Trabajando con imports</div>
  </body>
</html>
```

Como vemos, tendremos el siguiente error en consola:

```
Uncaught ReferenceError: hello is not defined
    at index.js:1
```

Esto se debe a que al cambiar el orden de importación, cuando se intenta realizar la llamada a *hello()* desde el fichero de *index.js*, esta función aún no existe en nuestro contexto global del documento, ya que aún no se ha realizado la importación del script *utils.js*

> ¿Qué ocurre si importamos un segundo fichero *utils2.js* con una función que también se llama *hello*?

Vamos a crear el siguiente fichero *utils2.js* para saludar con el texto *¡Hello!* 😃:

```
function hello() {
  return "Hello! :)";
}
```

Y lo importamos desde *index.html* en el siguiente orden:

```
<!DOCTYPE html>
<html lang="es">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0"
/>
    <meta http-equiv="X-UA-Compatible" content="ie=edge" />
    <title>Trabajando con imports</title>
    <script src="./utils.js"></script>
    <script src="./utils2.js"></script>
    <script src="./index.js"></script>
  </head>
  <body>
    <div>Bootcamp Javascript Lemoncode - Trabajando con imports</div>
  </body>
</html>
```

Como vemos en nuestro navegador, obtenemos como resultado que se está imprimiendo el texto de la función *hello* de nuestro segundo script (se ha reemplazado la función con el mismo nombre que existía en los imports anteriores).

> Con esto se evidencia los problemas que tendríamos trabajando con esta forma de importar funcionalidad, y la facilidad de incluir errores o efectos secundarios al llamar a funcionalidad con los mismos nombres en diferentes ficheros o al no especificar correctamente el orden en las importaciones de los scripts.

Trabajando con el objeto window para crear nuestra funcionalidad

Para poner un poco de concierto con todo lo anterior, una práctica que se solía utilizar era crear la funcionalidad de nuestra aplicación en el objeto *window* de del documento, creando namespaces en este objeto para organizarlo (App, Business, Utils, etc).

El documento *window* es global, y podemos ampliarlo con la funcionalidad que creamos necesaria para poder acceder a ella desde los diferentes ficheros de javascript.

Para realizar esto, utilizamos funciones autoinvocadas que reciben como parámetro el espacio del objeto *window* donde vamos a localizar la funcionalidad:

utils.js

```
(function(App) {
  App.hello = function() {
    return "¡Hola! :)";
  };
})(window.App || (window.App = {}));
```

index.js

```
(function(App) {
  const text = App.hello();
  document.write(text);
})(window.App || (window.App = {}));
```

La importación de los scripts sería similar a la sección anterior:

index.html

```
<!DOCTYPE html>
<html lang="es">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0"
/>
    <meta http-equiv="X-UA-Compatible" content="ie=edge" />
    <title>Trabajando con imports</title>
    <script src="./utils.js"></script>
    <script src="./index.js"></script>
  </head>
  <body>
    <div>Bootcamp Javascript Lemoncode - Trabajando con imports</div>
  </body>
</html>
```

Y nos encontramos con prácticamente los mismos problemas:

- Ensucia nuestro html.
- Hay que tener en cuenta el orden en el que importamos los scripts.
- Alta posibilidad de tener colisiones globales en nuestro documento con funcionalidad con el mismo nombre.

Solucionando el problema: Módulos de Javascript

Todo lo anterior es un problema conocido que se ha llevado muchos años en el desarrollo con javascript. Con el tiempo han ido surgiendo librerías para intentar solucionar el problema creando sistemas de módulos.

Podemos empezar a hablar del concepto de *Módulo* en javascript. Pero, un navegador por sí sólo no va a ser capaz de ejecutar nuestro código, ya que vamos a necesitar paquetes (librerías) externas, utilizar nuevas

versiones de javascript (ES6), etc. Por esto, vamos a realizar los ejercicios con *CodeSandbox*: CodeSandbox: Online Code Editor Tailored for Web Application Development, que nos ofrece un entorno de trabajo preparado para esto (podemos elegir el template de *Vanilla Javascript*). Como reto/extra, existe un documento en este módulo para preparar este entorno de trabajo en nuestro equipo local, creando un *Playground* con **NodeJS** y **Parcel** (que nos vaya sonando).

Librerías para trabajar con módulos

Todo lo anterior es un problema conocido que se ha llevado muchos años en el desarrollo con javascript. Con el tiempo han ido surgiendo librerías para intentar solucionar el problema creando sistemas de módulos, como *CommonJS*.

Trabajar con módulos de CommonJS

CommonJS es una librería para concertar los imports de funcionalidad y crear un sistema de módulos para facilitarnos la vida a la hora de trabajar con funcionalidades de diferentes ficheros y aliviar los problemas que hemos visto en las secciones anteriores.

No necesitaremos incluir en nuestro html todos los scripts que deseamos utilizar. En cada script importaremos los ficheros con la funcionalidad que deseamos utilizar.

Para importar ficheros utilizaremos *require(...)*:

src/index.js

```
const Utils = require("./utils");

document.write(Utils.hello());
```

Para exportar funcionalidad, utilizaremos *module.exports*:

src/utils.js

```
const hello = function() {
  return "¡Hola! :)";
};

module.exports = {
  hello: hello
};
```

Y en nuestro html importaríamos el punto de entrada a nuestra aplicación, en este caso *index.js*:

```
<html lang="es">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0"
```

```
  />
      <meta http-equiv="X-UA-Compatible" content="ie=edge" />
      <title>Trabajando con imports</title>
      <script src="./src/index.js"></script>
    </head>
    <body>
      <div>Bootcamp Javascript Lemoncode - Trabajando con imports</div>
    </body>
  </html>
```

> ¿Y si queremos utilizar la funcionalidad de *utils2* con el mismo nombre *hello*?

Vamos a crear primero el fichero *utils2* con la siguiente funcionalidad:

src/utils2.js:

```
const hello = function() {
  return "Hello! :)";
};

module.exports = {
  hello: hello
};
```

Ahora podemos importarlo desde *index.js*. Como ya hemos creado una constante con el nombre *Utils*, no nos va a permitir crear otra con el mismo nombre. Por tanto, vamos a importar la funcionalidad de *utils2* en una constante *Utils2*:

```
const Utils = require("./utils");
const Utils2 = require("./utils2");

document.write(Utils.hello());
document.write(Utils2.hello());
```

Como vemos, ya no tenemos colisión con las distintas funcionalidades de diferentes ficheros aunque se llamen igual, y no importa el orden en el que las importemos. Tenemos separados por módulos la funcionalidad de cada uno.

> Nota: El siguiente enlace a CodeSandbox tiene el ejemplo completo:
> https://codesandbox.io/s/bootcamp-lemoncode-mod8-content-3-7hx8o

Tenemos más sabores para exportar funcionalidad con *CommonJS*, utilizando el comando *export. [funcionalidad]*. Vamos a ver un ejemplo modificando la forma de exportar la funcionalidad de *utils2.js*:

src/utils2

```
const hello = function() {
  return "Hello! :)";
};

exports.hello = hello;
```

De esta forma podemos ir asignando cada una de las funcionalidades a propiedades dentro de nuestro namespace.

ES6: import y export

> Nota: igualmente necesitaremos utilizar un entorno de trabajo preparado para la ejecución. Podemos utilizar *CodeSandbox* para realizar las prácticas. El siguiente ejemplo está disponible en el siguiente enlace: https://codesandbox.io/s/bootcamp-lemoncode-mod8-content-4-uy5l8

Con la versión ES6 de javascript, se decidió dar solución a toda esta problemática desde el propio lenguaje, y además de una forma sencilla y fácil de utilizar.

El concepto es bastante parecido al ejemplo de *CommonJS*, y utilizaremos las palabras clave **import** y **export** para importar y exportar funcionalidad respectivamente.

Vamos a ver el ejemplo anterior, utilizando *import* y *export* de ES6:

src/utils.js

```
const hello = function() {
  return "¡Hola! :)";
};

export { hello };
```

src/utils2.js

```
const hello = function() {
  return "Hello! :)";
};

export { hello };
```

src/index.js

```
import * as Utils from "./utils";
import * as Utils2 from "./utils2";

document.write(Utils.hello());
document.write(Utils2.hello());
```

index.html

```
<html lang="es">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0"
/>
    <meta http-equiv="X-UA-Compatible" content="ie=edge" />
    <title>Trabajando con imports</title>
    <script src="./src/index.js"></script>
  </head>
  <body>
    <div>Bootcamp Javascript Lemoncode - Trabajando con imports</div>
    <div id="root"></div>
  </body>
</html>
```

Exportación por defecto

Podemos realizar exportaciones por defecto con el comando *export default*. Si utilizamos esta forma de exportar, podemos asignar el nombre que queramos a la funcionalidad que estamos exportando:

src/utils.js

```
const hello = function() {
  return "¡Hola! :)";
};

export default hello;
```

src/index.js

```
import helloFunction from "./utils";
document.write(helloFunction());
```