ES6 Modules

Los programas JavaScript comenzaron siendo bastante pequeños, la mayor parte de su uso en los primeros días era para realizar tareas de scripting aisladas, proporcionando un poco de interactividad a tus páginas web donde fuera necesario, por lo que generalmente no se necesitaban grandes scripts. Pasado los años los cambios son bastantes significativos, ahora tenemos aplicaciones completas que se ejecutan en navegadores con mucho JavaScript.

Por lo tanto, en los últimos años se ha comenzado a pensar en proporcionar mecanismos para dividir programas JavaScript en módulos separados que se puedan importar cuando sea necesario. Node.js ha tenido esta capacidad durante mucho tiempo, y hay una serie de bibliotecas y marcos de JavaScript que permiten el uso de módulos (por ejemplo, CommonJS y AMD otros basados en sistemas de módulos como RequireJS, y recientemente Webpack y Babel) y el día de hoy gracias a las mejoras constantes del lenguaje, los módulos son soportados de forma nativa y podemos utilizarlo sin ninguna dificultad.

Al momento de hacer uso de ES6 modules debemos tomar en cuanta algunos detalles:

- Debes prestar atención a las pruebas locales — si intentas cargar el archivo HTML localmente (es decir, con una URL file:///), te encontrarás con errores de CORS debido a los requisitos de seguridad del módulo JavaScript. Necesitas hacer tus pruebas a través de un servidor.
- Además, ten en cuenta que puedes obtener un comportamiento diferente de las secciones del script definidas dentro de los módulos en comparación con los scripts estándar. Esto se debe a que los módulos automáticamente usan strict mode.
- No es necesario utilizar el atributo defer (ve atributos de <script>) al cargar un script de módulo; los módulos se difieren automáticamente.
- Los módulos solo se ejecutan una vez, incluso si se les ha hecho referencia en varias etiquetas <script>.
- Por último, pero no menos importante, dejemos esto en claro las características del módulo se importan al alcance de un solo script no están disponibles en el alcance global. Por lo tanto, solo podrás acceder a las funciones importadas en el script en el que se importan y no podrás acceder a ellas desde la consola de JavaScript, por ejemplo. Seguirás recibiendo errores de sintaxis en DevTools, pero no podrás utilizar algunas de las técnicas de depuración que esperabas utilizar.

El último punto es algo de suma importancia dado que cualquier información que procesemos desde un módulo, solo es accesible desde ese modulo y no puede ser modificada por otros módulos o submódulos, amenos que estos estén siendo pasados como parámetros y/o gestionados por algún manejador de datos global. Ejemplo: el manejador de estados de REACT.

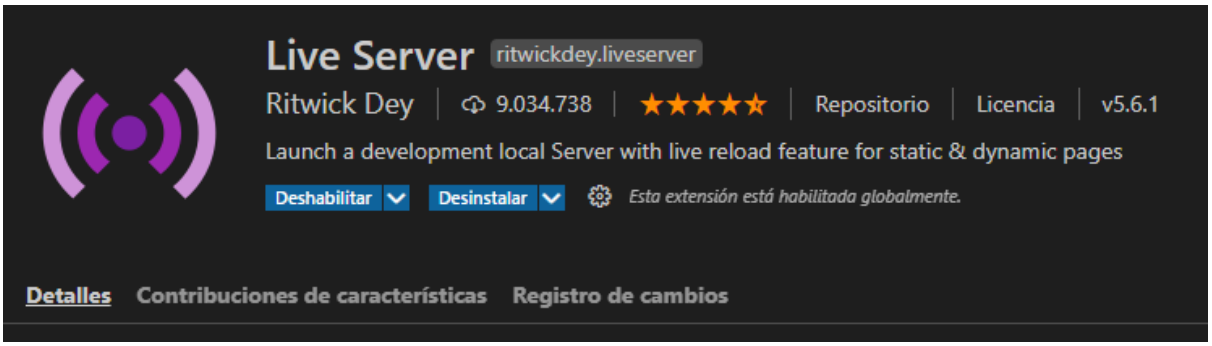
Con respecto al punto uno solo tendremos en cuenta que debemos ejecutar nuestra app desde un servidor ya sea local, externo o un servidor virtual.

Ejemplos de servidores que se pueden utilizar no requieren mucha configuración:

XAMPSERVER: este web server que incluye apache dentro de sus características, es bien fácil de utilizar, solo se debe instalar y colocar dentro de la carpeta htdocs la carpeta de nuestra app, luego acceder desde el “localhost/nombre_de_carpeta”, asumiendo que tienes

problemas de puerto, solo se debe configurar el xamp para usar un puerto determinado distinto al que está definido por defecto.

Live Server: esta es una extensión de VSCode permite correr aplicaciones en un server virtual fácil de utilizar.



Tome en cuenta que estas son solo un par de recomendaciones para hacer pruebas de su app, dado que hay muchos más server y extensiones que podrían fácilmente dar el mismo resultado, adicional a esto la mayoría de los IDE de desarrollo poseen su propio server virtual de pruebas. Y una vez que la app será puesta en producción los servers por defecto soportan ES6 modules.

IMPORT/EXPORT

Los ES6 modules traen consigo la potencia de hacer uso de la importación y exportación de librerías/bloques de código JS, permitiendo que las aplicaciones no tengan que cargar todos los elementos a la vez, así mismo la exportación puede ser tanto dinámica como estática, lo que implica que puedo declarar la traída de cada script desde el momento de la definición de la aplicación, o cargarlo en tiempo de ejecución por medio de acciones y eventos.

La declaración **export** se utiliza al crear módulos de JavaScript para exportar funciones, objetos o tipos de dato primitivos del módulo para que puedan ser utilizados por otros programas con la sentencia import. Los módulos exportados están en strict mode tanto si se declaran, así como si no. La sentencia export no puede ser utilizada en scripts embebidos.

Exportación por defecto: normalmente es utilizada cuando un módulo está diseñado solo para exportar un único elemento ya sea objeto, función o datos.

```
export default function myFun() {
      //code
}
```

Dado que myFun es una exportación por defecto, este puede ser invocada con cualquier nombre de instancia desde cualquier parte del código.

```
import mF from './ubicacion/myFuncModule.js'
```

Exportación de elementos específicos: esta se utiliza cuando un modulo posee múltiples objetos exportables.

```
function myFun() {}
class myClass {}
let myVar = "value";
export {myFun, myClass, myVar}
```

al hacer uso de este tipo de exportación, al momento de importar estos elementos se debe hacer uso del nombre de este para especificar cual es el que será utilizado.

```
import {myClass} from './myModule.js'
import {myFun} from './myModule.js'
```

También es posible hacer múltiples importaciones en la misma línea.

```
import {myClass, myVar} from './myModule.js'
```

o en el caso de querer exportar todos los elementos usar (este tipo de exportación no se puede usar para importar elemento con exportación por defecto):

```
import * from './myModule.js'
```

Para realizar exportaciones dinámicas, se debe hacer uso de promesas, por medio de la sintaxis async/await. Esta es una de las incorporaciones más recientes al lenguaje JavaScript, son las funciones asincrónicas, que permiten hacer uso de la palabra clave await, la cual permite indicarle a la función que debe esperar a que se resuelva una promesa, antes de continuar ejecutando el código.

La sintaxis sería bastante simple, se declararía la función que se encargaría de realizar la importación, esta sería una función del tipo asíncrona y la importación se realizaría por medio de la declaración de una variable, que este definida por la importación del módulo utilizando la palabra clave await:

```
const myFun = async() => {
    const { myClass } = await import ('./myModule.js');
    //continuar con el código
}
```

Es posible que nuestro modulo tenga solamente la definición de nuestro componente y no tenga en ningún momento definido ninguna exportación, por lo que no es posible declarar

el nombre de ningún objeto a importar, por lo que simplemente queremos que en cuanto el modulo se ponga en marcha defina algún customElement o simplemente ejecute ciertas tareas, para lograr esto simplemente debemos definir la importación.

De forma estática de la siguiente forma:

```
import "./myModule.js";
```

y de forma dinámica dentro de una función de la siguiente forma:

```
const myFun = async() => {
    await import('./myModule.js');
    //continuar con el código
}
```

Habiendo explicado todo esto, es momento de poner en práctica esta lógica dentro de nuestro entorno de trabajo. Para ello partiremos de modificar nuestro index.html, dado que necesitamos quitar las referencias a los módulos y de paso convertir el index.js en uno.

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
    <script src="./index.js" type="module"></script>
    <!-- referencias a estilos, scripts, coreTools, Components etc.  -->
</head>
<body id="App">
</body>
</html>
```

Como podemos notar hemos retirado las referencias a CardComponent y a WComponentsTools, por otro lado dentro del index js, para hacer uso de estos elementos haremos importación de estos objetos, por lo que el WComponentsTools debería poder exportar el WRender, y en el caso de CardComponent no será necesario hacer ningún tipo de exportación dado que solamente haremos uso de la definición de customElement.

Por lo que el WComponentsTools quedaría de la siguiente manera:

```
class WRender {
    static createElement = (Node) => {
        try {
            if (typeof Node === "undefined" || Node == null) {
                return document.createTextNode("Nodo nulo o indefinido.");
            } else if (typeof Node === "string" || typeof Node === "number") {
                return document.createTextNode(Node);
            } else if (Node.__proto__.__proto__ === HTMLElement.prototype) {
                return Node;
            } else {
                const element = document.createElement(Node.type);
                if (Node.props != undefined && Node.props.__proto__ == Object.prototype) {
                    for (const prop in Node.props) {
                        if (prop == "class") element.className = Node.props[prop];
                        else element[prop] = Node.props[prop];
                    }
                }
                if (Node.children != undefined && Node.children.__proto__ == Array.prototype) {
                    Node.children.forEach(Child => {
                        element.appendChild(this.createElement(Child));
                    });
                }
```

```
                return element;
            }
        } catch (error) {
            console.log(error, Node);
            return document.createTextNode("Problemas en la construcción del nodo.");
        }
    }
}
export { WRender }
```

El CardComponent necesitaría hacer uso de la importación del WRender para poder ejecutar sus métodos, por lo que incluiríamos en la cabecera de este la importación de esta clase import {WRender} from "../WModules/WComponentsTools.js"

```
import { WRender } from "../WModules/WComponentsTools.js";
class WCardComponent extends HTMLElement {
    constructor() {
        super();
        this.attachShadow({ mode: "open" });
    }
    connectedCallback() {
        this.shadowRoot.append(this.DrawCard());
    }
    DrawCard() {
        return WRender.createElement({
            type: "label",
            props: { className: "card" },
            children: [
                { type: "label", props: { className: "title", innerText: this.element.title } },
                { type: "section", props: { innerText: this.element.Contain } },
                { type: "section", props: { innerText: this.element.Detail } },
            ]
        });
    }
}
customElements.define("w-card", WCardComponent);
```

el index.js también necesitaría importar el WRender y la definición de nuestro componente, al utilizar importación estática quedaría de la siguiente forma.

```
import { WRender } from "./WDevCore/WModules/WComponentsTools.js";
import "./WDevCore/WComponents/CardComponent.js"
window.onload = () => {
    const cards = [
        { title: "Card 1", Contain: "Contain", Detail: "Detail" },
        { title: "Card 1", Contain: "Contain", Detail: "Detail" },
        { title: "Card 1", Contain: "Contain", Detail: "Detail" }
    ]
    const Frag = document.createDocumentFragment();
    for (let index = 0; index < cards.length; index++) {
        const element = cards[index];
        Frag.append(WRender.createElement({ type: "w-card", props: { element: element } }));
    }
    App.append(Frag);
}
```