ESPE

UNIVERSIDAD DE LAS FUERZAS ARMADAS

INNOVACIÓN PARA LA EXCELENCIA

"UNIVERSIDAD DE LAS FUERZAS ARMADAS"

ESPE

INGENIERÍA DE SOFTWARE

Programación web avanzada

TAREA #2

ESTUDIANTE:

Cristian Acalo

Ariel Guevara

César Loor

Erick Moreira

TUTOR ENCARGADO:

Ing. Doris Chicaiza

FECHA DE ENTREGA:

08 noviembre 2024

NRC:

2305

2. Objetivo General

Crear un aplicativo web que muestra como es el uso de funciones asincrónicas, Callback, Promesas, Async/ Await, con un ejemplo donde se simule su uso esto mostrándose en una página web.

3. Desarrollo

- **Conceptos base**:

Por defecto, JavaScript es un lenguaje de programación síncrono de un solo hilo. Esto significa que las instrucciones pueden solamente ejecutarse una después de la otra, y no en paralelo. Afortunadamente, los problemas con JavaScript síncrono fueron abarcados al introducir JavaScript asíncrono. Imagina el código asíncrono como código que comienza ahora, y termina su ejecución después. JavaScript está ejecutando asincrónicamente, las instrucciones no son necesariamente ejecutadas una después de la otra como vimos antes.

Un callback es una función que se pasa dentro de otra función, y luego se lo llama dentro de esa función para realizar una tarea.

Una promesa, en nuestro contexto, es algo que tomará algún tiempo en hacerse. Hay dos resultados posibles de una promesa:

- Ejecutamos y resolvemos la promesa
- Presenta un error a lo largo de la línea y la promesa es rechazada

Las promesas vinieron para resolver los problemas de las funciones callback. Una promesa toma dos funciones como parámetros. Eso es, resolve y reject. Recuerda que resolve es éxito, y reject es para cuando un error ocurre.

La cosa es, el encadenamiento de promesas juntos, justo como los callbacks se pueden poner bastante voluminoso y confuso. Por eso se produjo async y await.

La cosa es, el encadenamiento de promesas juntos, justo como los callbacks se pueden poner bastante voluminoso y confuso. Por eso se produjo async y await.

La palabra clave async es lo que usamos para definir las funciones asíncronas, por otro lado await impide a JavaScript de asignar fetch a la variable respuesta hasta que la promesa se haya resuelto. Una vez que la promesa ha sido resuelta, los resultados del método fetch pueden ahora ser asignados a la variable respuesta.

- **Herramientas de desarrollo**:
  IDE de trabajo Visual Studio Code
  JavaScript con el entorno de NodeJs

- **Cuerpo o Desarrollo**:
  - **Secciones o Subtítulos**: Estructura la información en secciones para organizar mejor el contenido. Cada sección debe abordar un aspecto específico de la tarea.
  - **Contenido**: Desarrolla cada punto de la tarea de forma clara y concisa. Utiliza párrafos bien estructurados y explica las ideas con ejemplos si es necesario.
  - **Apoyo visual** (si aplica):

```
console.log('aparece primero');
console.log('aparece segundo');

setTimeout(()=>{
    console.log('aparece tercero');
},2000);

console.log('aparece ultimo');
```

El fragmento de arriba es un pequeño programa que imprime cosas a la consola. Pero hay algo nuevo aquí. El interpretador ejecutará la primera instrucción, luego la segunda, pero salteará la tercera y ejecutará la última.

El setTimeout es una función de JavaScript que toma dos parámetros. El primer parámetro es otra función, y el segundo es el tiempo después el cual esa función debería ser ejecutada en milisegundos. Ahora ves la definición de callbacks entrando en juego.

```
const obtenerDatos = (dataEndpoint) => {
   return new Promise ((resolve, reject) => {
     //alguna peticion al endpoint;

     if(peticion es exitosa){
       //hace algo;
       resolve();
     }
     else if(hay un error){
       reject();
     }

   });
};
```

El código de arriba es una promesa, encerrado por una petición a algún endpoint. La promesa toma resolve y reject como se mencionó.

Después de hacer una llamada al endpoint por ejemplo, si la petición es exitosa, resolveríamos la promesa y continuaríamos haciendo lo que quisiéramos con la respuesta. Pero si hay un error, la promesa será rechazada.

```
const funcAsync = async () => {
    const respuesta = await fetch(recurso);
    const datos = await respuesta.json();
}
```

En el código de arriba tenemos un ejemplo de la implementación de una función asíncrona donde esta función async siempre regresará una promesa. En este caso el método .json regresa una promesa, y podemos usar await todavía para retrasar la asignación hasta que la promesa sea resuelta.

link Git Hub: https://github.com/ArielGuevara/WEB-Avanzada.git

4. Conclusión

La elección para usar estas herramientas depende del caso de uso, pero para la mayoría de los proyectos modernos, async/await se ha convertido en el estándar debido a su claridad y facilidad de uso, mientras que las promesas siguen siendo fundamentales en operaciones más complejas y los callbacks se limitan a escenarios específicos. De tal modo que, la combinación adecuada de estas herramientas permite construir aplicaciones más eficientes, organizadas y mantenibles.

5. Referencias

- *Introducción a JavaScript asíncrono - Aprende desarrollo web | MDN*. (s. f.). MDN Web Docs. https://developer.mozilla.org/es/docs/Learn/JavaScript/Asynchronous/Introducing

- *Cabrera, F. (2022, 18 de enero). JavaScript asincrónico: explicación de callbacks, promesas y async/await. FreeCodeCamp Español. Recuperado de https://www.freecodecamp.org/espanol/news/javascript-asincrono-explicacion-de-callbacks-promesas-y-async-await/*

6. Anexo

https://github.com/CesarLoor/WebAvanzada_2305.git