Técnico en DESARROLLO DE SOFTWARE

Programación Avanzada

Programación Avanzada

Unidad III

1. ¿Qué es la programación asincrónica?

La programación como la venimos viendo está pensada de principio a fin como un solo proceso, es decir, si tenemos alguna función o alguna llamada a un API o una base de datos esperamos a que esta termine antes de continuar con la ejecución del programa.

¿Qué problema tiene eso? imaginemos una consulta a una base de datos, está solamente analiza una muestra de 10 filas, esta consulta se resolverá en cuestión de segundos.

Ahora imaginemos la misma consulta, solo que se hace a una base de datos con 10 millones de filas. Esta toma más tiempo, pero posiblemente no lo suficiente para crear un problema.

Ahora imaginemos que lo que estamos solicitando no son filas de una base de datos, sino imágenes o videos.

Las consultas de texto pesan a lo mucho algunos kb, pero las imágenes y videos pueden llegar a pesar en términos de GB.

¿Y qué pasa mientras descargamos imágenes o videos? todo lo demás se va a quedar esperando, y puede que veamos la conocida ventana de “Esta aplicación no responde”.

Pensemos en un navegador web, eso no es lo que sucede, mientras descargamos archivos podemos seguir navegando, un proceso no acapara todos los recursos, es posible tener varios procesos ejecutándose al mismo tiempo.

Pensemos en otro caso alejado de las computadoras, un pedido en un restaurante, cuando llegamos lo primero es ver el menú, luego hacer un pedido, esta ira a la cocina, ahí se prepara y luego nos lo entregan.

Todos estos son pasos independientes, y se realizan asincrónicamente. Imaginemos que sea sincrónico: Una vez que se coloca un pedido, no podemos poner otro, es decir se atiende a un cliente a la vez, entonces tenemos que esperar todo el proceso de pedido > preparación > entrega. Es como si solamente un empleado estuviera a cargo de todo el restaurante.

En realidad, sabemos que esto no es así. Una vez que se colocan las órdenes se empieza a preparar la comida, pero el empleado que está tomando órdenes vuelve con los clientes en espera y sigue tomando órdenes, Deja “pendiente” la preparación de la comida y regresa con el cliente una vez que ya está lista. No se queda esperando a que salga la comida y deja esperando a los clientes

Esta forma de llevar a cabo los procesos se llama programación asincrónica, los procesos se inician y detienen en tiempos diferentes y podemos seguir realizando otras cosas en paralelo a los procesos existentes.

La programación asincrónica es la forma de programar no secuencial, es decir que no esperamos a que algo termine, sino que seguimos con la ejecución de nuestro programa.

Un problema de esto es que los resultados de los procesos que llamamos pueden ser veloces, o pueden ser lentos, entonces tenemos que tener el cuidado de no tratar de usar los resultados antes de que estén listos. Por ejemplo, una descarga de una imagen, si queremos compartirla antes de que termine la descarga la imagen estará incompleta o corrupta. Por esta razón los lenguajes de programación modernos no nos dan la opción de ver los resultados de un proceso asincrónico, en lugar de esto, los procesos nos avisan de cualquier progreso, error o finalización. En otras palabras un proceso asincrónico nos dice "No nos llame, nosotros lo llamaremos".

JavaScript es un lenguaje asincrónico por diseñó, muchas de las funciones que existen en JS son asincrónicas.

2. Callbacks

En JS una función de callback es una función que recibe como argumento otra función, sin embargo más comúnmente se utiliza para continuar el funcionamiento de un código.

Entonces podríamos tener una función de esta forma

```
function funcion1(argumento, callback){
    callback(argumento)
}
```

Esta función recibe 2 parámetros, 1 argumento y 1 función, luego manda a llamar la función con el argumento.

La función que estamos pasando puede ser cualquier función por ejemplo

```
function mostrarSaludo(usuario){
    alert('Hola '+usuario)
}
```

Y luego lo podemos llamar así

```
function funcion1(argumento, callback){
  callback(argumento)
}

function mostrarSaludo(usuario){
  alert('Hola '+usuario)
}

funcion1('usuario', mostrarSaludo)
```

Entonces lo que estamos haciendo es crear una función que llama a otra función, este comportamiento ya lo hemos visto anteriormente. Cuando tratamos con los listeners para los eventos de los botones u otros componentes de una página web.

Un listener se ve similar a esto.

```
var boton = document.getElementById('boton1')


boton.addEventListener('click', funcionBoton1)


function funcionBoton1(){
  alert('Hiciste click en el botón')
}
```

¿Por qué hacer uso de funciones de callback?

Se ve claro cuando trabajamos con los listeners, JS es un lenguaje dirigido por eventos y asincrónico, los eventos ocurren en tiempos diferentes cada vez que se ejecuta un programa y no son solamente clics o escritura en el teclado, los eventos pueden ser solicitudes de red, ingreso de datos, etc. Entonces las funciones de callback nos permiten trabajar con una función cuando ocurre un evento y nos permite verificar las condiciones en las que ocurrió, por ejemplo, verificar si el usuario ha leído y aceptado los términos de uso de la aplicación antes de crear una cuenta. También podemos verificar si el usuario tiene conexión a internet, si es una cuenta válida, etc.

Una función de callback, a su vez, puede llamar otra función de Callback, entonces, por ejemplo, podemos tener una función que obtiene un parámetro de consulta, esta función verifica que sea un parámetro válido, y lo pasa a otra función, que hace una consulta a una base de datos, esta función no hace nada con el resultado, lo pasa a otra función que comprueba la validez de la respuesta, y solamente cuando la respuesta es válida lo devolvemos al usuario.

Uno de los callback más conocidos de JS es setTimeout, es una función que nos permite colocar un tiempo de espera, similar al sleep o wait de otros lenguajes de programación. La diferencia es que setTimeout tiene una función de callback, por lo que incluye el código que vamos a ejecutar a la hora de terminar la espera.

```
setTimeout ( ()=>{
    alert("Han pasado 3 segundos")
} ,3000)
```

Esta función se puede usar para simular eventos de espera cuando aún estamos en fases de prueba.

3. Promesas

Las promesas surgen como una forma de realizar callback con una sintaxis más legible y ordenada. ¿Por qué no seguir con los callbacks? ¿Cuál es el problema? El problema no es con los callbacks, sino con la forma en la que se tienen que manejar, un solo callback no es un problema, pero cuando tenemos 2 o más, se empieza a crear un código muy difícil de leer, esto se le conoce como *Callback Hell.*

Más información sobre el callback hell y porque es un problema: http://callbackhell.com

Las promesas resuelven esto creando una estructura que, en lugar de colocar código dentro de la función, lo separa dentro de una llamada a la función *then()*, esto permite que el código se pueda leer más claramente.

Las promesas son funciones que "prometen" retornar un resultado. Esto significa que siempre nos van a avisar si la operación es exitosa o no, a diferencia de los callbacks, en los que decimos que vamos a hacer con el resultado obtenido, las promesas simplemente nos regresan el resultado para que pueda ser usado.

Una promesa se crea como una función con dos parámetros, resolve y reject, estas son funciones de JavaScript que nos permiten avisar el resultado de nuestra función, si fue exitosa enviamos un resolve con el resultado, si tuvo un error enviamos un reject con el error.

Una promesa se ve así

```
var promesa = new Promise ( function(resolve, reject){
        //hacemos algo en la promesa
        if( esExitoso){
            resolve("Valor de éxito")
        }else{
            reject(Error("Hubo un error"))
       }
}
```

Declarar una función de esta forma puede que no se le vea mucha utilidad ¿de qué sirve una función a la que no le podemos pasar parámetros?, lo que podemos hacer es declarar promesas dentro de funciones y así crear funciones asincrónicas.

```
function validarUsuario(nombre){
    return new Promise(  (resolve, reject) =>{
        var valido = false
        //validaciones de usuario en las que puede cambiar el valor de valido
        if(valido){
            resolve("Usuario Válido")
        }else{
            reject(Error("Usuario Invalido"))
        }
    })
}
```

Luego esto se puede llamar como promesa de esta forma

```
validarUsuario()
    .then(res => console.log(res))
    .catch(err => console.log(err))
```

Veamos cómo funciona esta parte:

Primero está la llamada a la función, recordemos colocar los paréntesis, se colocan cuando queremos llamar la función, cuando la estamos usando como parámetro (como en los callbacks) no se usa.

Luego vemos la función then, recordemos que then se usa para trabajar cuando una promesa ha acabado. Desde el then podemos acceder a los valores de resolve declarando una función que maneje el mismo, en este caso estamos usando la notación de flecha, pero también podemos hacer uso de la declaración de una función normal con la palabra clave function.

La función dentro del then recibe el resultado en forma de la variable que colocamos, luego podemos manejar el resultado de la promesa.

Después del then, podemos colocar 2 cosas, si la función que declaramos es otra Promise, podemos colocar otro then que manejará el resultado de esa promesa.

Si ya no tenemos más promesas, podemos colocar un catch, la función de catch se encarga de manejar el resultado de reject de una promesa.

Por ejemplo, si el usuario es invalido

En la parte de reject de una Promesa podemos retornar un valor así como en el resolve. Pero estamos retornando un Error para que nos muestre las líneas del programa en el que sucedió el error.

4. Async / Await

Una de las nuevas características de JS son las funciones async, básicamente son funciones que siempre van a retornar una promesa.

La palabra clave await nos permite esperar a que la función asíncrona retorne.

Son básicamente otra forma de colocar una Promesa y un then

La forma de usar estas características es simple, para declarar una función async se hace con la palabra clave async de esta forma

```
async function funcionAsincronica(){
    //hacemos algo de manera asincrónica en la función
}
```

Una vez que declaramos una función async podemos esperar a que termine su ejecución con la palabra clave await de la siguiente forma

```
await funcionAsincronica()
```

Un detalle importante es que solamente podemos hacer un await dentro de una función async. Como se mencionó anteriormente, esto es igual que hacer uso de una Promesa y luego colocar un then, nos servirá cuando tengamos varias operaciones que realizar y cada una necesite del resultado del paso anterior.

Veamos cómo se vería si quisieramos simular la validación del usuario que vimos anteriormente, pero esta vez agregando un retrason con setTimeout y usando una función async

```
function validarUsuario(nombre){
    return new Promise(  (resolve, reject) =>{
        setTimeout(()=>{
            var valido = true
            //validaciones de usuario en las que puede cambiar el valor de valido
            if(valido){
                resolve("Usuario Valido")
            }else{
                reject(Error("Usuario Invalido"))
            }
        }, 3000)
    })
}

async function simularEspera(){
    let x = await validarUsuario("usuario")
    alert(x)
}
simularEspera()
```

Es la misma función, pero ahora tenemos, dentro de la promesa, un setTimeout, que nos hace esperar 3 segundos.

Veamos la función simularEspera, tenemos una asignación de la función validar usuario. Si hiciéramos esto sin la palabra await, la variable x tendría de valor una promesa, con la palabra clave await vamos a tener el valor que retorne la promesa.

La siguiente línea es un alert. Este nos va a mostrar un valor en pantalla. Si no usáramos la palabra clave await, entonces nos mostraría que x es un objeto Promesa, sin embargo, al usar await, la ejecución del programa se vuelve sincrónica y espera a que se resuelva la promesa.

Con la palabra await, entonces el alert nos va a mostrar el valor resultante de la promesa.

```
async function simularEspera(){
    let x = validarUsuario("usuario")
    alert(x)
}
```

Aquí el alert nos mostrara lo siguiente

Y lo mostrará de manera inmediata, entonces no está esperando a que se resuelva el timeout.

Siempre hay que recordar que async/await es solamente otra forma de declarar promesas, y una función async lo que hace es retornar una promesa, entonces, dejando todo igual en la función de validar nombre, y cambiando un poco la de simularEspera, podemos tratar el resultado como una promesa sin usar la palabra clave await

```
async function simularEspera(){
    return validarUsuario("usuario")
}

simularEspera().then(alert)
```

Esto tendrá el mismo resultado que trabajar con el await

Descargo de responsabilidad

La información contenida en este documento descargable en formato PDF o PPT es un reflejo del material virtual presentado en la versión online del curso. Por lo tanto, su contenido, gráficos, links de consulta, acotaciones y comentarios son responsabilidad exclusiva de su(s) respectivo(s) autor(es) por lo que su contenido no compromete al área de e-Learning del Departamento GES o al programa académico al que pertenece.

El área de e-Learning no asume ninguna responsabilidad por la actualidad, exactitud, obligaciones de derechos de autor, integridad o calidad de los contenidos proporcionados y se aclara que la utilización de este descargable se encuentra limitada de manera expresa para los propósitos educacionales del curso.