ASINCRONISMO EN JAVASCRIPT

TEMA 04 – CALLBACKS

4.1. Introducción a los callbacks

En JavaScript, gran parte de la potencia y flexibilidad del lenguaje radica en su capacidad para trabajar de forma asíncrona, es decir, para llevar a cabo operaciones en segundo plano sin bloquear la ejecución principal de un programa. Una de las primeras y más tradicionales formas de gestionar estas operaciones asíncronas en JavaScript es mediante el uso de *callbacks*.

A lo largo de esta sección, abordaremos en profundidad qué son los callbacks, cómo funcionan en el lenguaje y cuáles son los inconvenientes que pueden surgir al utilizarlos de manera excesivamente anidada, fenómeno conocido como *Callback Hell*.

4.2. ¿Qué son los callbacks y cómo funcionan?

Un *callback* es, en esencia, una función que se pasa como argumento a otra función para que esta la ejecute más tarde, ya sea después de completar una tarea asíncrona o en respuesta a un evento específico. Dicho de otra manera, un callback sirve para indicarle a una función qué hacer una vez que haya terminado una cierta operación (por ejemplo, una llamada a un servidor, la lectura de un archivo, o un temporizador).

En JavaScript, las funciones se pueden:

1. **Asignar** a una variable.
2. **Pasar** como argumento a otra función.
3. **Devolver** desde una función.

Esta característica del lenguaje facilita la creación y el uso de funciones de callback. El patrón típico de uso es:

1. Se define una función principal que realiza una operación (por ejemplo, solicitar datos a un servidor).

2. Se pasa como argumento una segunda función (callback) para que la primera la invoque una vez que termine su tarea.

Veamos un ejemplo sencillo que muestra cómo funcionan las callbacks:

```
// Función que simula una tarea asíncrona, por ejemplo, consulta a una API
function getDataFromServer(callback) {
  console.log("Iniciando la petición al servidor...");
  // Usamos un setTimeout para simular un retardo
  setTimeout(function() {
    const data = { nombre: "Juan", edad: 30 };
    console.log("Datos obtenidos del servidor.");
    // Llamamos a la función de callback con los datos
    callback(data);
  }, 2000);
}
// Función que se utilizará como callback para manejar los datos
function processData(data) {
  console.log("Procesando los datos recibidos...");
  console.log(`Nombre: ${data.nombre}, Edad: ${data.edad}`);
}
// Invocamos la función principal y le pasamos 'processData' como callback
getDataFromServer(processData);
```

En este ejemplo:

• getDataFromServer(callback) simula una operación asíncrona que, en la vida real, podría ser una petición a un servidor externo.

• Dentro de setTimeout, pasado cierto tiempo, se obtienen supuestamente unos datos.

• Una vez disponibles los datos, la función callback (en este caso processData) se invoca y recibe esos datos como argumento.

• processData a su vez muestra la información por consola.

Orden de ejecución

Un detalle crucial para entender cómo funcionan los callbacks es el orden en el que se ejecutan. JavaScript, a pesar de ser un lenguaje monohilo, cuenta con un *Event Loop* que gestiona las operaciones asíncronas. Cuando setTimeout finaliza el tiempo de espera, la función de callback que se le ha pasado se coloca en la cola de tareas (Task Queue).

En cuanto el *Event Loop* detecta que la pila de ejecución está vacía, transfiere esa función de la cola a la pila para que finalmente se ejecute. Este proceso explica por qué la parte sincrónica del código (instrucciones que no dependen de la operación asíncrona) se ejecuta primero, mientras que la ejecución de la función callback se realiza después.

Uso de callbacks en eventos

Otra faceta muy habitual de los callbacks en JavaScript se da en la interacción con el DOM (Documento de Modelo de Objetos) y la gestión de eventos. Por ejemplo, se puede asignar una función callback que se dispare al hacer clic en un botón:

```
<button id="miBoton">Haz clic aquí</button>
<script>
  const boton = document.getElementById("miBoton");
  // Definimos la función callback que se ejecutará tras el clic
  function manejarClick() {
    alert("¡Has hecho clic en el botón!");
  }
  // Añadimos la función callback como listener del evento 'click'
  boton.addEventListener("click", manejarClick);
</script>
```

Este tipo de patrones es muy común en la programación web, donde se delega a las callbacks la responsabilidad de reaccionar a eventos externos como clics de ratón, pulsaciones de teclado, envío de formularios, etc.

4.3. Problemas del “Callback Hell”

La versatilidad de los callbacks hizo que durante mucho tiempo fueran la principal forma de controlar la asincronía en JavaScript. Sin embargo, a medida que las aplicaciones crecieron en complejidad, se popularizó un término para describir la excesiva anidación de funciones callback: el llamado *Callback Hell*.

¿Qué es el Callback Hell?

El *Callback Hell* ocurre cuando, para llevar a cabo una secuencia de operaciones asíncronas, vamos encadenando múltiples callbacks de manera anidada. El resultado es un código que se ve cada vez más inclinado hacia la derecha, difícil de leer y, sobre todo, muy complicado de mantener o depurar. Un ejemplo muy caricaturesco de *Callback Hell* puede lucir así:

```
funcionAsincrona1(function(resultado1) {
  funcionAsincrona2(resultado1, function(resultado2) {
    funcionAsincrona3(resultado2, function(resultado3) {
      funcionAsincrona4(resultado3, function(resultado4) {
        // ...
        console.log("Demasiado anidado y difícil de seguir...");
      });
    });
  });
});
```

Aunque este ejemplo es intencionadamente exagerado, es frecuente encontrarse con estructuras similares en proyectos grandes si no se siguen algunas pautas de organización del código.

Por qué es problemático

Legibilidad reducida:

A medida que se añaden más callbacks, el código se va moviendo hacia la derecha y su estructura se vuelve más compleja y menos intuitiva. Esto dificulta entender de un vistazo qué está ocurriendo y en qué orden.

Mantenimiento complicado:

Con muchos niveles de anidación, cualquier cambio en la lógica (por ejemplo, añadir una nueva condición o una verificación de errores en algún punto intermedio) puede requerir reestructurar por completo el bloque de callbacks.

Gestión de errores confusa:

En ocasiones, la información de error debe pasar a través de múltiples funciones para tratar cada situación. Controlar de manera ordenada cada posible fallo puede volverse muy engorroso cuando cada nivel de callback debe gestionar excepciones o devolver información de error.

Buenas prácticas para evitar el Callback Hell

Aunque la sección se enfoca en los callbacks, es relevante mencionar algunas pautas que tradicionalmente se han usado para mantener el código limpio y evitar caer en el *Callback Hell*:

Modularizar las funciones:

En lugar de definir la lógica completa dentro de un único callback, se puede dividir en funciones más pequeñas, cada una encargada de una parte de la operación, y luego encadenarlas con nombres descriptivos.

Manejar errores de forma centralizada:

Definir una convención clara para manejar errores en los callbacks. Por ejemplo, la convención node.js de callback(error, data), donde el primer parámetro es el posible error y el segundo son los datos de la operación, ayuda a estandarizar la gestión de fallos.

Usar Promesas y posteriormente async/await:

Mediante el uso de promesas y el de async/await, el código asíncrono se ha vuelto más lineal y más fácil de manejar. Aunque estos temas se abordan en secciones dedicadas, conviene destacar que, en la práctica

actual, se recomienda migrar o arrancar los proyectos directamente con Promesas o con async/await, dejando los callbacks para casos puntuales.

En conclusión, los callbacks constituyen la base histórica del manejo de asincronía en JavaScript. Son una herramienta potente, pero el abuso o la ausencia de patrones claros al emplearlos puede derivar en *Callback Hell*, dificultando la legibilidad y el mantenimiento de nuestro código. Por ello, es fundamental conocer sus ventajas y desventajas, y sobre todo, estar familiarizados con las mejoras y evoluciones que ha tenido el lenguaje para manejar la asincronía de manera más elegante y limpia (como Promesas o async/await). Conociendo estos conceptos, podremos crear aplicaciones JavaScript más robustas y fáciles de mantener.

4.4. Ejemplos

Ejemplo 1: callback sincrónico

Aunque la esencia de los callbacks en JavaScript está muy ligada a la asincronía, es posible utilizar esta técnica de manera “sincrónica” para fines didácticos. En este ejemplo, simplemente pasamos una función como parámetro a otra función y la ejecutamos inmediatamente:

```
/** Función que recibe un callback y lo ejecuta inmediatamente
 * con un mensaje. */
function ejecutaCallback(callback) {
  const mensaje = "Saludo desde ejecutaCallback";
  callback(mensaje);
}
// Definimos el callback que recibirá el mensaje
function mostrarMensaje(texto) {
  console.log("Callback ejecutado: " + texto);
}
// Llamamos a la función principal pasando 'mostrarMensaje' como
callback
ejecutaCallback(mostrarMensaje);
```

Explicación:

1. La función ejecutaCallback genera un mensaje en una variable local.

2. Invoca el callback mostrarMensaje pasándole ese mensaje.

3. El callback se ejecuta y muestra el texto por consola.

Este ejemplo es puramente ilustrativo y se centra en mostrar el paso de funciones como argumentos, sin la parte asíncrona tan típica de JavaScript.

Ejemplo 2: callback asíncrono

En este caso, aprovechamos la función setTimeout para simular una llamada asíncrona. Una vez transcurrido el tiempo especificado, se invoca la función de callback:

```
/** Función que simula un proceso asíncrono con setTimeout.
 * Recibe un callback que se ejecutará después de 2 segundos. */
function procesoAsincrono(callback) {
  console.log("Proceso iniciado...");
  setTimeout(function() {
    const resultado = "Datos obtenidos tras un tiempo de espera";
    // Invocamos el callback pasándole el resultado
    callback(resultado);
  }, 2000);
}
// Definimos el callback para manejar el resultado
function manejarResultado(dato) {
  console.log("Callback manejando el resultado: " + dato);
}
// Llamamos a la función principal y pasamos nuestro callback
procesoAsincrono(manejarResultado);
```

Explicación

1. console.log("Proceso iniciado...") se ejecuta primero.

2. Después de 2 segundos, manejarResultado se invoca con el mensaje "Datos obtenidos tras un tiempo de espera".

3. Este patrón refleja la base del manejo asíncrono con callbacks en JavaScript.

Ejemplo 3: callback hell

El *Callback Hell* se produce cuando tenemos una serie de operaciones asíncronas que dependen unas de otras y las encadenamos de manera excesivamente anidada. A continuación, se muestra un ejemplo intencionadamente exagerado para ilustrar cómo se ve un código difícil de mantener:

```
// Simulamos funciones asíncronas anidadas
function operacion1(datos, callback) {
  setTimeout(function() {
    console.log("Operación 1 completada con:", datos);
    callback(datos + " -> O1");
  }, 1000);
}
function operacion2(datos, callback) {
  setTimeout(function() {
    console.log("Operación 2 completada con:", datos);
    callback(datos + " -> O2");
  }, 1000);
}
function operacion3(datos, callback) {
  setTimeout(function() {
    console.log("Operación 3 completada con:", datos);
    callback(datos + " -> O3");
  }, 1000);
}

// Aquí vemos la anidación que provoca el 'Callback Hell'
operacion1("Iniciando", function(resultado1) {
  operacion2(resultado1, function(resultado2) {
```

```
    operacion3(resultado2, function(resultado3) {
      console.log("Resultado final tras múltiples operaciones: " +
resultado3);
      // Podríamos seguir anidando más y más...
    });
  });
});
```

Problemas que se encuentran:

Excesiva indentación: El código se desplaza gradualmente hacia la derecha, dificultando la legibilidad.

Dificultad de mantenimiento: Si quisiéramos modificar alguna parte del flujo o controlar errores en cada paso, se complicaría notablemente.

Escalabilidad limitada: Añadir más pasos requiere anidar más callbacks, empeorando la situación.

Ahora vamos a ver dos ejemplos más completos donde combinamos tanto la parte teórica de los *callbacks* como la posibilidad de caer en un escenario de *Callback Hell*. El objetivo es mostrar cómo interactúan varias funciones asíncronas que comparten o transforman datos.

Ejemplo 4: Envío de datos a un servidor y notificación al usuario

Supongamos que tenemos estas tareas:

1. Recoger datos de un formulario.
2. Enviarlos al servidor (simulado con setTimeout).
3. Mostrar una notificación tras el envío.

```
/** Simula la recolección de datos de un formulario. Como podría
ser rápido o lento, se gestiona asíncronamente con setTimeout. */
function recogerDatosFormulario(callback) {
  console.log("Recogiendo datos del formulario...");
```

```
/** Suponemos un retardo de 1s para simular que puede tardar en completarse */
  setTimeout(function() {
    const datos = {
      nombre: "Laura",
      email: "laura@example.com"
    };
    console.log("Datos del formulario obtenidos.");
    callback(datos);
  }, 1000);
}
/** Simula el envío de datos a un servidor. Cuando finaliza, invoca el callback. */
function enviarDatosAServidor(datos, callback) {
     console.log(`Enviando los datos de ${datos.nombre} al servidor...`);
  // Simulamos un retardo de 2s
  setTimeout(function() {
             console.log(`Datos de ${datos.nombre} enviados correctamente.`);
    callback();
  }, 2000);
}
// Simula la notificación al usuario tras completar el envío.
function notificarUsuario() {
     console.log("Notificación: El proceso ha finalizado con éxito.");
}
// Encadenamos las llamadas utilizando callbacks
recogerDatosFormulario(function(datosRecibidos) {
  enviarDatosAServidor(datosRecibidos, function() {
    notificarUsuario();
  });
});
```

Explicación:

1. recogerDatosFormulario recoge (de manera simulada) los datos e invoca el callback con dichos datos.

2. En la función anónima que recibe datosRecibidos, se llama a enviarDatosAServidor, pasándole datosRecibidos y otra función de callback.

3. Tras completar el envío, se llama a notificarUsuario.

Este flujo no es muy extenso, pero si añadimos más pasos asíncronos (validación, transformaciones, consultas adicionales, etc.), se incrementaría la probabilidad de caer en un *Callback Hell*.

Ejemplo 5: Cadena de validaciones con anidación

En este segundo ejemplo, mostraremos cómo varias validaciones consecutivas pueden incrementar la complejidad del código si seguimos usando solo callbacks sin una estructura clara.

```
// Validación 1: Comprobar que el usuario no está bloqueado
function checkUserNotBlocked(usuario, callback) {
  setTimeout(function() {
    if (usuario.bloqueado) {
      return callback("Usuario bloqueado, no puede continuar.");
    }
    console.log("Validación 1: El usuario no está bloqueado.");
    callback(null, usuario);
  }, 1000);
}
// Validación 2: Comprobar que el usuario tiene permisos
function checkUserPermissions(usuario, callback) {
  setTimeout(function() {
    if (!usuario.permisos.includes("ADMIN")) {
```

```
      return callback("El usuario no tiene permisos suficientes.");
    }
    console.log("Validación 2: El usuario posee los permisos necesarios.");
    callback(null, usuario);
  }, 1000);
}
// Validación 3: Comprobar que la suscripción está activa
function checkSubscription(usuario, callback) {
  setTimeout(function() {
    if (!usuario.suscripcionActiva) {
      return callback("La suscripción del usuario no está activa.");
    }
    console.log("Validación 3: La suscripción del usuario está activa.");
    callback(null, usuario);
  }, 1000);
}
// Simulamos un objeto 'usuario'
const usuarioEjemplo = {
  nombre: "Carlos",
  bloqueado: false,
  permisos: ["ADMIN", "EDITOR"],
  suscripcionActiva: true
};
/** Llamadas encadenadas: Ejemplo que puede complicarse conforme crezcan las validaciones */
checkUserNotBlocked(usuarioEjemplo, function(error, usuario1) {
  if (error) {
    return console.error("Error:", error);
  }
  checkUserPermissions(usuario1, function(error, usuario2) {
    if (error) {
      return console.error("Error:", error);
```

```
  }
  checkSubscription(usuario2, function(error, usuario3) {
    if (error) {
      return console.error("Error:", error);
    }
    console.log(
      "Todas las validaciones pasadas con éxito para el usuario:",
      usuario3.nombre
    );
  });
 });
});
```

Explicación:

1. Cada validación simula un proceso asíncrono (con setTimeout).

2. Si hay un error, se retorna inmediatamente para interrumpir el flujo.

3. En caso de éxito, se pasa el usuario al siguiente callback hasta completar todas las validaciones.

4. A medida que se van añadiendo más verificaciones, el anidamiento crece y complica la lectura, ilustrando la problemática típica del *Callback Hell*.

4.5. Conclusión

Estos ejemplos reflejan las bondades y los desafíos de los *callbacks*. Han sido la base para manejar la asincronía en JavaScript durante mucho tiempo, pero, como hemos visto, pueden derivar en un *Callback Hell* que resulte poco mantenible. Por eso, en la actualidad se recomienda hacer uso de promesas y, especialmente, de la sintaxis async/await para escribir código más legible, aunque conocer los callbacks sigue siendo esencial para comprender el modelo de ejecución de JavaScript y mantener proyectos que aún los utilicen.

Es fundamental el conocimiento de estos patrones de programación asíncrona, tanto en su versión con callbacks como en la posterior evolución

con Promesas y async/await, con el fin de adquirir una comprensión completa del asincronismo en JavaScript.