ASINCRONISMO EN JAVASCRIPT

TEMA 09 – ASYNC / AWAIT

9.1. Introducción

Para situarnos en contexto, es importante matizar por qué `async/await` existe en JavaScript y en qué se basa. Cuando hablamos de programación asíncrona en JavaScript, hay varios momentos clave:

1. **Callbacks**: fueron la forma tradicional de manejar operaciones asíncronas. Sin embargo, a medida que los programas crecían en complejidad, se hacía habitual encontrarnos con estructuras de "callback hell", anidando funciones y generando código difícil de leer y de mantener.
2. **Promesas**: introducidas para solucionar los problemas de las funciones de callback anidadas. Las Promesas permiten encadenar métodos como `then()` y `catch()`, lo cual aporta un mayor orden y manejabilidad a las operaciones asíncronas. Aun así, en proyectos grandes, una larga sucesión de `then()` encadenados puede llegar a ser confusa y costosa de leer.
3. **Async/Await**: surge como una evolución o *azúcar sintáctico* sobre las Promesas, incorporado formalmente a partir de ECMAScript 2017. El término ***azúcar sintáctico*** significa que no se trata de un mecanismo nuevo, sino que internamente sigue basándose en el comportamiento de las Promesas, pero ofrece una forma de escribir el código más clara, parecida a la programación secuencial tradicional.

En pocas palabras, `async/await` no reemplaza a las Promesas, sino que las utiliza para manejar los procesos asíncronos. Gracias a esto, el código asíncrono se aproxima más al estilo sincrónico de siempre (un paso tras otro, en orden), lo que favorece la legibilidad y reduce los posibles errores lógicos.

¿Por qué async/await se considera más legible?

- ***Menor anidación***: los bloques `then()` se sustituyen por un código que se ve más secuencial.
- ***Manejo de errores más coherente***: en lugar de depender de métodos específicos como `.catch()`, se utiliza `try/catch` nativo de JavaScript, que resulta más familiar y consistente.
- ***Lectura lineal***: la palabra clave `await` "pausa" la ejecución de la función hasta que la Promesa se resuelve (o se rechaza), por lo que el flujo se lee de manera lineal, como si fuese código síncrono.

De esta forma, `async/await` es como una capa sobre las Promesas, ya que internamente no deja de usarlas, pero nos ofrece un modo más sencillo e intuitivo de trabajar con ellas.

9.2. Uso de async/await

Para empezar a usar `async/await`, debemos entender claramente las dos palabras clave involucradas: **async** y **await**.

- `async`: se antepone a la declaración de una función (por ejemplo, `async function miFuncion() { ... }`), indicando que dicha función va a contener código asíncrono y que, además, retornará automáticamente una Promesa.
- `await`: solo puede usarse dentro de una función declarada como `async`. Su función principal es "esperar" a que una Promesa se resuelva o se rechace, como si pausara la ejecución hasta que el resultado esté disponible. Durante ese tiempo de espera, la ejecución del resto del código dentro de esa función queda en suspenso. Eso sí, mientras tanto, JavaScript puede continuar ejecutando otros trozos de código en el programa principal (event loop), lo que mantiene el carácter asíncrono del lenguaje.

```
// Simulando la obtención de datos de un servidor con un retardo usando setTimeout
function obtenerDatosServidor() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      // Imaginemos que los datos llegan correctamente
      const datos = { usuario: "Juan", id: 123 };
      resolve(datos);
    }, 2000); // 2 segundos de retardo
  });
}

async function procesarDatos() {
  console.log("Iniciando solicitud de datos...");
  // Esperamos a que la Promesa se resuelva
  const datos = await obtenerDatosServidor();
  console.log("Datos recibidos:", datos);
  console.log("Continuamos con el flujo de ejecución...");
}

procesarDatos();
// Salida en consola:
// "Iniciando solicitud de datos..."
// (Tras 2 segundos) "Datos recibidos: { usuario: 'Juan', id: 123 }"
// "Continuamos con el flujo de ejecución..."
```

Observa que `await` no bloquea todo el programa, sino solo el bloque de código dentro de la función `async`. Fuera de esa función, el resto del código JavaScript puede continuar ejecutándose (por ejemplo, eventos del navegador, otras funciones, etc.). Sin embargo, dentro de la función `procesarDatos()`, la línea con `await obtenerDatosServidor()` actúa como un “alto” hasta que la Promesa se cumpla o se rechace.

Este paradigma hace que el código sea más lineal y fácil de entender en comparación con el encadenamiento de `then()` y `catch()`, o el uso intensivo de callbacks.

9.3. Manejo de errores con try/catch

Cuando usábamos Promesas con el enfoque clásico, disponíamos del método `.catch()` para capturar los errores. Con `async/await`, se adopta la sintaxis clásica de JavaScript para gestionar excepciones: `try/catch`.

La idea es envolver las llamadas asíncronas que podrían fallar en un bloque `try/catch`, capturando así cualquier error que la Promesa lance en caso de que sea rechazada.

```
async function obtenerUsuario() {
  // Supongamos que esta función retorna una promesa con datos de un usuario
  return { nombre: "Pedro", edad: 30 };
}

async function mostrarUsuario() {
  try {
    // Si la promesa falla, se lanza un error que será capturado por el catch
    const usuario = await obtenerUsuario();
    console.log("Usuario obtenido:", usuario);
  } catch (error) {
    console.error("Ocurrió un error al obtener el usuario:", error);
  }
}
mostrarUsuario();
```

Si `obtenerUsuario()` rechazara la Promesa (por ejemplo, usando `reject` en la lógica interna), el bloque `catch` dentro de `mostrarUsuario()` recibiría el error y podríamos manejarlo apropiadamente.

```
function obtenerDetallesProducto(idProducto) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      // Imaginemos que si el idProducto es negativo o cero, es un ID inválido
      if (idProducto > 0) {
```

```
      resolve({ id: idProducto, nombre: "Teclado", precio: 30 });
    } else {
      reject(new Error("ID de producto inválido"));
    }
  }, 1000);
 });
}

async function procesarProducto(idProducto) {
 try {
  const producto = await obtenerDetallesProducto(idProducto);
  console.log("Detalles del producto:", producto);
 } catch (error) {
  console.error("Error al procesar producto:", error.message);
 }
}

procesarProducto(10); // Caso válido
procesarProducto(-1); // Caso inválido
```

En el caso válido, tras un segundo se imprime:

```
Detalles del producto: { id: 10, nombre: "Teclado", precio: 30 }
```

En el caso inválido, el mensaje de error se capturará y mostrará en consola:

```
Error al procesar producto: ID de producto inválido
```

Gracias a `try/catch`, el tratamiento de errores en `async/await` se asemeja a lo que se haría en un lenguaje de programación tradicional. Esto reduce la necesidad de anidar múltiples `.catch()` en diferentes puntos, lo que agiliza la lectura y hace el flujo de trabajo más claro.

9.4. EJEMPLOS

EJEMPLO 1

Este primer ejemplo muestra cómo se podría escribir una función asíncrona con Promesas y luego la misma función usando `async/await`, para apreciar que `async/await` es, en esencia, *azúcar sintáctico* sobre las Promesas.

```
// Función que simula la obtención de datos de un servidor
function obtenerDatosConPromesa() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const datos = { usuario: "Carlos", edad: 25 };
      // Simulamos que la operación se resuelve correctamente
      resolve(datos);
    }, 1000);
  });
}

// Uso de la función
obtenerDatosConPromesa()
  .then((resultado) => {
    console.log("Versión con Promesas - Datos recibidos:", resultado);
  })
  .catch((error) => {
    console.error("Versión con Promesas - Error:", error);
  });
```

Versión con Async/Await

```
// Función que simula la obtención de datos de un servidor, versión async/await
async function obtenerDatosAsyncAwait() {
  // Internamente, esta función sigue usando Promesas, pero se escribe de forma más legible
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const datos = { usuario: "Carlos", edad: 25 };
      resolve(datos);
```

```
    }, 1000);
  });
}

async function mostrarDatos() {
  try {
    const resultado = await obtenerDatosAsyncAwait();
    console.log("Versión con Async/Await - Datos recibidos:", resultado);
  } catch (error) {
    console.error("Versión con Async/Await - Error:", error);
  }
}

// Llamamos a la función principal para ver el resultado en consola
mostrarDatos();
```

En la primera parte, se emplea `.then()` y `.catch()` para gestionar la Promesa. En la segunda, se usa `await` y un bloque `try/catch`. Ambos métodos hacen lo mismo, pero la versión `async/await` suele leerse de manera más secuencial.

EJEMPLO 2

Para este ejemplo, tenemos una función que simula la consulta de un listado de productos.

```
// Función que simula la obtención de un listado de productos
function obtenerProductos() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const productos = [
        { id: 1, nombre: "Teclado", precio: 30 },
        { id: 2, nombre: "Ratón", precio: 15 },
        { id: 3, nombre: "Monitor", precio: 150 }
      ];
      resolve(productos);
    }, 1500);
```

```
  });
}

async function listarProductos() {
  console.log("Solicitando productos...");

  // El flujo de esta función se “detiene” aquí hasta que se resuelve la Promesa
  const lista = await obtenerProductos();

  console.log("Productos recibidos:");
  lista.forEach((producto) => {
    console.log(`- ${producto.nombre} (ID: ${producto.id}): ${producto.precio}€`);
  });

  console.log("Continuando la ejecución tras haber recibido la lista...");
}

// Llamamos a la función asíncrona
listarProductos();
```

EJEMPLO 3

En este ejemplo, veremos cómo capturar los errores de una Promesa rechazada dentro de una función marcada como *async*.

```
// Función que simula obtener datos de un usuario, pudiendo fallar
function obtenerUsuarioPorId(id) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      // Simulamos un fallo si el ID es <= 0
      if (id > 0) {
        resolve({ id: id, nombre: "María", rol: "Admin" });
      } else {
        reject(new Error("ID de usuario inválido"));
      }
    }, 1000);
  });
```

```
}

async function mostrarUsuario(id) {
  try {
    const usuario = await obtenerUsuarioPorId(id);
    console.log("Usuario obtenido:", usuario);
  } catch (error) {
    // Si la Promesa se rechaza, se captura aquí
    console.error("Error al obtener el usuario:", error.message);
  }
}

// Probar con un ID válido
mostrarUsuario(2);

// Probar con un ID inválido (lanzará la excepción en consola)
mostrarUsuario(0);
```

En este caso, se emplea el bloque `try/catch` para atrapar cualquier error que se produzca dentro de la función asíncrona. Si la Promesa se rechaza, el flujo salta al bloque `catch`, donde podemos manejar el error de la forma que necesitemos (mostrar un mensaje al usuario, reintentar la operación, etc.).

EJEMPLO 4

Supongamos que tenemos dos operaciones asíncronas: una para obtener la información de un usuario y otra para obtener sus pedidos (o compras) de una tienda en línea. Queremos unirlo todo de forma clara y legible:

```
// Función que obtiene información de un usuario
function obtenerInfoUsuario(idUsuario) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (idUsuario > 0) {
        resolve({ id: idUsuario, nombre: "Sofía" });
```

```
    } else {
      reject(new Error("ID de usuario no válido"));
    }
  }, 1000);
 });
}

// Función que obtiene la lista de pedidos de un usuario
function obtenerPedidosDeUsuario(idUsuario) {
 return new Promise((resolve, reject) => {
  setTimeout(() => {
    // Imaginamos que para este usuario existe una lista de pedidos
    const pedidos = [
      { numero: 101, total: 56 },
      { numero: 102, total: 120 },
    ];
    // Si no hay errores, resolvemos
    resolve(pedidos);
  }, 1200);
 });
}

async function mostrarDatosCompletosUsuario(idUsuario) {
 try {
  console.log("Iniciando obtención de datos...");

  // 1. Obtenemos la info básica del usuario
  const usuario = await obtenerInfoUsuario(idUsuario);
  console.log("Usuario encontrado:", usuario);

  // 2. Obtenemos los pedidos asociados a ese usuario
  const pedidos = await obtenerPedidosDeUsuario(usuario.id);
  console.log("Pedidos del usuario:", pedidos);

  console.log("Proceso finalizado con éxito.");
 } catch (error) {
```

```
    // Capturamos cualquier error de las dos funciones
    console.error("Error en la obtención de datos:", error.message);
  }
}

// Llamamos con un ID correcto
mostrarDatosCompletosUsuario(5);

// Llamamos con un ID incorrecto (<= 0), provoca error
mostrarDatosCompletosUsuario(-1);
```

En este ejemplo, se observa cómo:

1. La estructura secuencial (primero obtener el usuario, después los pedidos, etc.) es más fácil de entender.
2. El manejo de errores centralizado se realiza con `try/catch`, por lo que si falla la primera o la segunda llamada, el `catch` "atrapa" la excepción.

EJEMPLO 5

En este ejemplo, además de hacer dos solicitudes asíncronas, aprovechamos la sintaxis de `Promise.all()` junto a `async/await`. Esta combinación puede resultar útil cuando dos (o más) operaciones se pueden ejecutar en paralelo y queremos esperar a que todas finalicen antes de continuar.

```
// Función que simula obtener información de un curso
function obtenerInfoCurso(idCurso) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (idCurso > 0) {
        resolve({ id: idCurso, titulo: "Curso de JavaScript Avanzado" });
      } else {
        reject(new Error("ID de curso inválido"));
      }
    }, 1000);
  });
}
```

```
// Función que simula obtener el listado de alumnos de un curso
function obtenerAlumnosDelCurso(idCurso) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (idCurso > 0) {
        resolve(["Ana", "Luis", "Marta", "Pablo"]);
      } else {
        reject(new Error("No se pudieron obtener alumnos para este curso"));
      }
    }, 1500);
  });
}

async function cargarDatosCurso(idCurso) {
  try {
    console.log("Iniciando carga de datos del curso...");

    // Hacemos ambas llamadas en paralelo para ganar eficiencia
    const [curso, alumnos] = await Promise.all([
      obtenerInfoCurso(idCurso),
      obtenerAlumnosDelCurso(idCurso),
    ]);

    // Mostramos la información recibida
    console.log("Información del curso:", curso);
    console.log("Listado de alumnos:", alumnos);

    console.log("Datos del curso cargados con éxito.");
  } catch (error) {
    // Si ocurre algún error en cualquiera de las dos operaciones, se captura aquí
    console.error("Error al cargar los datos del curso:", error.message);
  }
}

// Llamada con un ID válido
```

```
cargarDatosCurso(10);


// Llamada con un ID inválido
cargarDatosCurso(0);
```

En este ejemplo:

1. Hemos utilizado `Promise.all()` para ejecutar ambas promesas en paralelo.
2. Gracias a `await`, esperamos a que ambas finalicen antes de asignar los resultados a las variables `curso` y `alumnos`.
3. Cualquier error que ocurra en una de las dos promesas se captura en el bloque `catch`.

9.5. CONCLUSIÓN

La llegada de `async/await` a JavaScript supuso un hito a la hora de trabajar con código asíncrono, simplificando la legibilidad y el manejo de excepciones. Aunque parezca una novedad radical, internamente se trata de una forma más “dulce” (o *azúcar sintáctico*) de usar Promesas. El resultado es un código más lineal y, por tanto, más fácil de mantener y depurar.

A modo de resumen, conviene retener los siguientes puntos clave:

1. **Async/Await se basa en Promesas**: no las elimina ni las sustituye, simplemente hace su uso más sencillo.
2. **Lectura secuencial**: el uso de `await` en una función `async` promueve que el código se lea como si fuese sincrónico.
3. **Manejo de errores**: mediante `try/catch` podemos capturar fácilmente los posibles errores que surjan en las operaciones asíncronas.
4. Comprender `async/await` puede ser más rápido que adaptarse a varios niveles de `then` y `callbacks`, ya que se aproxima más a la forma de pensar de la programación secuencial.