26. Ejemplo con Promesas

Usar **promesas** es una excelente manera de manejar asincronía en JavaScript, y es especialmente útil para evitar el "callback hell" (el anidamiento excesivo de callbacks). Además, las promesas son más intuitivas y fáciles de entender para muchos desarrolladores.

Vamos a reorganizar el código del ejemplo anterior para usar promesas en lugar de callbacks. Esto incluirá:

1. Modificar `baseDatos` para que devuelva una promesa.
2. Manejar la promesa en el archivo HTML usando `.then()` o `async/await`.
3. Mantener el mismo comportamiento: los números se renderizan en la tabla después de 6 segundos.

---

1. Modificar `baseDatos.js` para usar promesas

En lugar de aceptar un callback, `baseDatos` ahora devuelve una promesa que se resuelve con los resultados de la consulta.

`baseDatos.js`

```
const consulta = function (entrada) {
  return new Promise((resolve) => {
    let listado = [];
    if (entrada == "pares") {
      listado = Array.from({ length: 10 }, (_, i) => i + 2).filter(
        (i) => i % 2 === 0
      );
    } else {
      listado = Array.from({ length: 10 }, (_, i) => i + 1).filter(
        (i) => i % 2 === 1
      );
    }
    resolve(listado); // Resolvemos la promesa con los resultados
  });
};

export const baseDatos = function (peticion) {
  console.log("Petición recibida " + peticion);

  return new Promise((resolve) => {
    setTimeout(() => {
      consulta(peticion).then((listado) => {
        console.log("Consulta completada:", listado);
        resolve(listado); // Resolvemos la promesa con los resultados
      });
    }, 6000);
```

```
  });
};
```

2. Actualizar el archivo HTML para manejar la promesa

En el archivo HTML, usaremos `.then()` o `async/await` para manejar la promesa devuelta por `baseDatos`. Ambos enfoques son válidos, pero aquí te muestro ambos para que puedas elegir el que prefieras.

Opción 1: Usando `.then()`

```
<script type="module">
  import { baseDatos } from "./js/baseDatos.js";

  function actualizarTabla(datos) {
    const tabla = document.getElementById("tabla");
    tabla.innerHTML = ""; // Limpiar la tabla

    datos.forEach((numero) => {
      const fila = document.createElement("tr");
      const celda = document.createElement("td");
      celda.textContent = numero;
      fila.appendChild(celda);
      tabla.appendChild(fila);
    });
  }

  document.getElementById("disparador").addEventListener("click", () => {
    let eleccion = document.getElementById("pares_o_nones").value;

    baseDatos(eleccion).then((listado) => {
      console.log("Datos recibidos:", listado);
      actualizarTabla(listado); // Actualizamos la tabla con los datos
    });

    console.log("Petición realizada");
    console.log("Voy haciendo otras cosas");
  });
</script>
```

Opción 2: Usando `async/await`

```
<script type="module">
  import { baseDatos } from "./js/baseDatos.js";

  function actualizarTabla(datos) {
    const tabla = document.getElementById("tabla");
    tabla.innerHTML = ""; // Limpiar la tabla
```

```
  datos.forEach((numero) => {
    const fila = document.createElement("tr");
    const celda = document.createElement("td");
    celda.textContent = numero;
    fila.appendChild(celda);
    tabla.appendChild(fila);
  });
}

document
    .getElementById("disparador")
    .addEventListener("click", async () => {
      let eleccion = document.getElementById("pares_o_nones").value;

      // Iniciamos la petición a la base de datos
      const promesa = baseDatos(eleccion);
      // Guardamos la promesa sin esperarla todavía

      // Mostramos los mensajes después de iniciar la petición
      console.log("Petición realizada");
      console.log("Voy haciendo otras cosas");

      try {
        // Esperamos a que la promesa se resuelva
        const listado = await promesa;
        console.log("Datos recibidos:", listado);
        actualizarTabla(listado); // Actualizamos la tabla con los datos
      } catch (error) {
        console.error("Error al obtener los datos:", error);
      }
    });
</script>
```

3. Explicación del flujo con promesas

El flujo de trabajo con promesas es muy similar al de los callbacks, pero más limpio y estructurado:

1. El usuario selecciona "Pares" o "Impares" en el `<select>` y hace clic en el botón.
2. Al hacer clic, se llama a `baseDatos`, que devuelve una promesa.
3. La función `baseDatos` simula una consulta asincrónica durante 6 segundos.
4. Una vez completada la consulta, la promesa se resuelve con los resultados.
5. En el archivo HTML, manejamos la promesa usando `.then()` o `async/await`:
    - Si usamos `.then()`, pasamos una función que actualiza la tabla cuando la promesa se resuelve.

- Si usamos `async/await`, esperamos a que la promesa se resuelva y luego actualizamos la tabla.

---

4. Ventajas de usar promesas

1. **Código más limpio**: Las promesas evitan el anidamiento excesivo de callbacks, lo que mejora la legibilidad.
2. **Manejo de errores**: Las promesas permiten manejar errores de manera centralizada con `.catch()` o `try/catch` (en el caso de `async/await`).
3. **Compatibilidad moderna**: Las promesas son ampliamente utilizadas en APIs modernas de JavaScript, como `fetch`.

Código completo con promesas

baseDatos.js

```
const consulta = function (entrada) {
  return new Promise((resolve) => {
    let listado = [];
    if (entrada == "pares") {
      listado = Array.from({ length: 10 }, (_, i) => i + 2).filter(
        (i) => i % 2 === 0
      );
    } else {
      listado = Array.from({ length: 10 }, (_, i) => i + 1).filter(
        (i) => i % 2 === 1
      );
    }
    resolve(listado);
  });
};

export const baseDatos = function (peticion) {
  console.log("Petición recibida " + peticion);

  return new Promise((resolve) => {
    setTimeout(() => {
      consulta(peticion).then((listado) => {
        console.log("Consulta completada:", listado);
        resolve(listado);
      });
    }, 6000);
  });
};
```

Asincronia.html

```html
<!DOCTYPE html>
<html lang="en">
 <head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>Promesas</title>
 </head>
 <body>
  <select name="" id="pares_o_nones">
   <option value="pares">Pares</option>
   <option value="nones">Impares</option>
  </select>
  <button id="disparador">Llama BaseDatos</button>
  <table id="tabla" border="1"></table>

  <script type="module">
   import { baseDatos } from "./js/baseDatos.js";

   function actualizarTabla(datos) {
    const tabla = document.getElementById("tabla");
    tabla.innerHTML = "";

    datos.forEach((numero) => {
     const fila = document.createElement("tr");
     const celda = document.createElement("td");
     celda.textContent = numero;
     fila.appendChild(celda);
     tabla.appendChild(fila);
    });
   }

   document.getElementById("disparador").addEventListener("click", async () => {
    let eleccion = document.getElementById("pares_o_nones").value;

    console.log("Petición realizada");
    console.log("Voy haciendo otras cosas");

    try {
     const listado = await baseDatos(eleccion);
     console.log("Datos recibidos:", listado);
     actualizarTabla(listado);
    } catch (error) {
     console.error("Error al obtener los datos:", error);
    }
   });
  </script>
```

```
  </body>
</html>
```

Conclusión

Usar promesas es una excelente manera de manejar asincronía en JavaScript. Es más moderno, limpio y fácil de entender que los callbacks tradicionales. Además, enseñar promesas a tus alumnos les preparará para trabajar con APIs modernas como `fetch`, que también usan promesas.

Con esta implementación, tus alumnos podrán ver cómo las promesas simplifican el manejo de operaciones asincrónicas y cómo `async/await` puede hacer que el código sea aún más legible. ¡Espero que esto sea útil para tus clases! 😊