ASINCRONISMO EN JAVASCRIPT

TEMA 10 – WEB WORKERS

10.1. Introducción a los Web Workers

Hasta ahora hemos tratado el asincronismo desde la perspectiva clásica de JavaScript, es decir, un único hilo de ejecución junto con un segundo plano al que se envían las tareas asíncronas. HTML5 introdujo los **Web Workers**, cuyo objetivo principal es permitir la ejecución de scripts de manera **concurrente**, es decir, en hilo de ejecución separados y, de esta forma, liberar al hilo principal de la carga pesada, haciendo que la experiencia de usuario sea más fluida.

A continuación, veremos en detalle qué son los Web Workers, cómo crearlos y utilizarlos, y finalmente cuáles son sus limitaciones y buenas prácticas para sacar el máximo provecho de ellos.

10.2. Qué son y para qué sirven los Web Workers

Un Web Worker es un script de JavaScript que se ejecuta en un hilo distinto al principal con el objetivo de realizar tareas complejas o prolongadas. De esta manera:

- **El hilo principal** se encarga de gestionar la interfaz de la aplicación, la interacción con el DOM y la respuesta a eventos, manteniendo la aplicación receptiva en todo momento.
- **Los Web Workers** ejecutan operaciones intensivas de CPU sin interrumpir el flujo de la interfaz gráfica. Un ejemplo muy común es cuando necesitamos realizar cálculos matemáticos complejos, manipular grandes volúmenes de datos o procesar ficheros pesados.

La gran ventaja de los Web Workers es su capacidad de ejecutar código en paralelo. Cada Web Worker tiene su propio contexto global y no

comparte el objeto window del hilo principal, lo cual fomenta la independencia entre el cómputo intensivo y las funciones de actualización de la UI.

10.3. Diferencias entre asincronismo (Event Loop) y paralelismo (Web Workers)

Para entender mejor la importancia de los Web Workers, hay que distinguir dos conceptos clave:

1. **Asincronismo mediante el Event Loop:**

JavaScript, por naturaleza, es un lenguaje de ejecución **monohilo**. Todas las operaciones que realiza en el hilo principal se orquestan a través del **Event Loop**. En este modelo, las tareas se encolan (por ejemplo, con setTimeout, setInterval, peticiones AJAX, *promises*, etc.) y se ejecutan secuencialmente cuando el hilo principal está disponible. Las funciones asíncronas permiten que la interfaz de usuario no se bloquee mientras esperamos la resolución de procesos de E/S (entrada/salida), como peticiones de red, pero **no** aprovechan varios núcleos de CPU para ejecutar tareas en paralelo, sino que siguen gestionándose en un único hilo principal.

2. **Paralelismo con Web Workers:**

Los **Workers** permiten correr código de JavaScript en hilos separados del hilo principal, haciendo uso de varios núcleos de la CPU si están disponibles en el sistema. Esto significa que si una tarea compleja se está ejecutando en un Worker, el hilo principal puede seguir respondiendo a eventos del usuario, actualizar la interfaz, o realizar otras operaciones. En pocas palabras, con los Web Workers se consigue un verdadero **paralelismo**, siempre dentro de los límites del navegador y del modelo de hilos que ofrece el sistema operativo.

10.4. Creación de un Web Worker

La creación y uso de un Web Worker es relativamente sencilla, pero requiere prestar atención a algunos detalles específicos en cuanto a la comunicación entre el hilo principal y el propio Web Worker.

Comunicación entre el hilo principal y los Workers

La comunicación entre un Web Worker y el hilo principal se lleva a cabo a través de **mensajes**, usando los métodos y eventos:

1. `.postMessage()`

   Permite enviar información desde el hilo principal al Web Worker y viceversa.

2. `.onmessage()`

   Es un manejador de eventos que se dispara cuando uno de los dos extremos recibe un mensaje. Permite capturar y procesar la información que llega.

Estructura básica en el hilo principal

```
// 1. Creación de la instancia del Worker a partir de un archivo JS externo
const worker = new Worker('miWorker.js');
// 2. Envío de mensajes al Worker
worker.postMessage('Hola, Worker!');
// 3. Recepción de mensajes desde el Worker
worker.onmessage = function (event) {
  console.log('Mensaje recibido desde Worker:', event.data);
};
```

Estructura básica Web Worker (por ejemplo, miWorker.js)

```
// 1. Recepción de mensajes desde el hilo principal
onmessage = function (event) {
  console.log('Mensaje recibido desde el hilo principal:', event.data);
  // 2. Podemos procesar la información y enviar la respuesta
```

```
  const respuesta = `He recibido tu mensaje: ${event.data}`;
  postMessage(respuesta);
};
```

El intercambio de datos se basa en un mecanismo de **paso de mensajes**. No existe un acceso directo desde el Worker al objeto window o al DOM del hilo principal, lo cual es una de las restricciones más notables, pero también forma parte de la filosofía de aislar procesos y evitar bloqueos de la interfaz.

10.5. Limitaciones y buenas prácticas

Aunque los Web Workers son muy útiles, no están exentos de limitaciones. Para aprovecharlos de manera óptima, conviene tener presentes ciertos aspectos:

1. **Acceso limitado al DOM**

Dentro de un Worker no tienes acceso directo al DOM ni al objeto document o window. Esta limitación existe para evitar bloqueos en la interfaz y garantizar la independencia del hilo de ejecución. Si necesitas actualizar la interfaz, debes comunicarte con el hilo principal usando postMessage y dejar que éste se encargue de manipular el DOM.

2. **No se pueden usar ciertas funciones del navegador**

Muchas APIs y métodos que dependen del objeto window o de la interfaz gráfica no están disponibles, por ejemplo, no puedes llamar a `alert()`, `prompt()` o usar `document.createElement()` directamente dentro de un Worker. Sin embargo, sí puedes usar gran parte de las API web como `fetch()` y similares, siempre y cuando no requieran acceso directo al DOM.

3. **Sobrecarga de creación y comunicación**

Crear un Worker y comunicarse con él puede acarrear un pequeño coste adicional. Por ello, no es recomendable crear excesivos Workers para tareas muy pequeñas. Es preferible reservarlos para procesos que de verdad lo requieran, como transformaciones de grandes volúmenes de datos o cálculos matemáticos intensivos.

Además, cada Worker consume recursos (memoria y CPU). Un número muy elevado de Workers podría saturar el sistema.

4. **Estructura de archivos**

Los Workers deben ser definidos en archivos JavaScript separados e instanciarse con `new Worker('archivoWorker.js')`. No se puede “incrustar” el script de un Worker en una misma página HTML usando el mismo contexto, lo que obliga a separar el código y mantener una organización clara de ficheros.

5. **Sincronización y concurrencia**

Aunque los Workers proporcionan paralelismo, aún tenemos que pensar en la sincronización. Si varios Workers manipulan los mismos datos (por ejemplo, acceden a una misma base de datos externa), debemos coordinar esas operaciones de forma que no se produzcan inconsistencias.

6. **Utiliza técnicas de optimización**

Al transmitir datos muy grandes (arrays, objetos), considera usar `transferables` o técnicas de serialización eficientes para minimizar la sobrecarga de copia de datos.

Emplea algoritmos bien diseñados y evita recursividad extrema sin necesidad.

7. **Manejo adecuado de errores**

Manejar los eventos `onerror` en el Worker y en el hilo principal para detectar cualquier falla en la ejecución del script.

Recuerda que las excepciones que suceden dentro de un Worker no bloquearán el hilo principal, por lo que es importante llevar un registro de posibles errores.

10.6. EJEMPLOS

EJEMPLO 1: EVENT LOOP VS WEB WORKER

Código asíncrono tradicional (sin paralelismo real):

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8" />
  <title>Asincronía - Event Loop</title>
</head>
<body>
  <h2>Ejemplo de Asincronía con Event Loop</h2>
  <button id="btnIniciar">Iniciar Tareas</button>
  <div id="output"></div>

  <script>
    const boton = document.getElementById('btnIniciar');
    const salida = document.getElementById('output');

    boton.addEventListener('click', () => {
      salida.innerHTML = '';
      salida.innerHTML += '<p>Inicio de la operación (hilo principal)</p>';

      // Tarea asíncrona: se ejecutará después de 2 segundos
      setTimeout(() => {
        salida.innerHTML += '<p>Tarea asíncrona completada tras 2 segundos</p>';
      }, 2000);

      // Tarea sincrónica inmediata
```

```
    salida.innerHTML += '<p>Fin del click (hilo principal)</p>';
  });
 </script>
</body>
</html>
```

Código con Web Worker (paralelismo real):

index.hmtl

```
<!DOCTYPE html>
<html>
<head>
 <meta charset="UTF-8" />
 <title>Comunicación con Web Worker</title>
</head>
<body>
 <h2>Comunicación Básica con un Web Worker</h2>
 <button id="btnEnviar">Enviar Mensaje al Worker</button>
 <div id="respuesta"></div>
 <script>
  // 1. Crear la instancia de Worker
  const worker = new Worker('workerBasico.js');
  const boton = document.getElementById('btnEnviar');
  const respuestaDiv = document.getElementById('respuesta');
  // 2. Evento para enviar mensaje al Worker
  boton.addEventListener('click', () => {
   worker.postMessage('Hola, Worker!');
  });
  // 3. Escuchar respuestas del Worker
  worker.onmessage = (event) => {
   respuestaDiv.textContent = `Respuesta del Worker: ${event.data}`;
  };
 </script>
</body>
</html>
```

workerBasico.js

```
// Escuchar el evento onmessage para recibir mensajes del hilo principal
onmessage = function(event) {
  console.log('Mensaje recibido del hilo principal:', event.data);

  // Responder al hilo principal
  const respuesta = `Worker ha recibido tu mensaje: "${event.data}"`;
  postMessage(respuesta);
};
```

EJEMPLO 2: FIBONACCI CON WEB WORKER

Este ejemplo demuestra el proceso de realizar una operación costosa (cálculo recursivo de Fibonacci) en un Worker, evitando que la interfaz se bloquee.

fibonacci.html

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8" />
  <title>Fibonacci con Worker</title>
</head>
<body>
  <h2>Cálculo de Fibonacci usando Web Worker</h2>
  <label for="numeroFibo">Número de Fibonacci a calcular:</label>
  <input type="number" id="numeroFibo" value="40" />
  <button id="btnCalcular">Calcular</button>
  <div id="resultado"></div>
  <div id="estado"></div>
  <script>
    const input = document.getElementById('numeroFibo');
    const boton = document.getElementById('btnCalcular');
    const resultado = document.getElementById('resultado');
    const estado = document.getElementById('estado');
```

```
    // Creación del Worker
    const workerFibo = new Worker('workerFibonacci.js');
    // Escuchar mensajes del Worker (resultado)
    workerFibo.onmessage = (event) => {
      resultado.textContent = `Resultado: ${event.data}`;
      estado.textContent = 'Cálculo finalizado.';
    };
    // Manejo de posibles errores dentro del Worker
    workerFibo.onerror = (event) => {
      estado.textContent = `Error en Worker: ${event.message}`;
    };
    // Evento del botón para enviar petición de cálculo
    boton.addEventListener('click', () => {
      estado.textContent = 'Calculando...';
      resultado.textContent = '';
      const valor = parseInt(input.value, 10);
      workerFibo.postMessage(valor);
    });
  </script>
</body>
</html>
```

workerFibonacci.js

```
// Cálculo recursivo de Fibonacci
function fibonacci(n) {
  if (n <= 1) return n;
  return fibonacci(n - 1) + fibonacci(n - 2);
}
onmessage = function(event) {
  const numero = event.data;
  const resultado = fibonacci(numero);
  postMessage(resultado);
};
```

EJEMPLO 3: LIMITACIONES Y BUENAS PRÁCTICAS

A continuación se muestra que **no se puede acceder al DOM** desde el Worker y cómo pasar datos más grandes mediante mensajes. La idea es ilustrar buenas prácticas y limitaciones:

limitaciones.html

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8" />
  <title>Limitaciones y Buenas Prácticas con Workers</title>
</head>
<body>
  <h2>Limitaciones de Web Workers</h2>
  <button id="btnProcesar">Procesar Lista en Worker</button>
  <div id="output"></div>
  <script>
    const workerLim = new Worker('workerLimitaciones.js');
    const boton = document.getElementById('btnProcesar');
    const salida = document.getElementById('output');
    workerLim.onmessage = (event) => {
      // Recibimos un array procesado y lo mostramos
      const arrayProcesado = event.data;
      salida.textContent = `El Worker devolvió un array de longitud: ${arrayProcesado.length}`;
    };
    boton.addEventListener('click', () => {
      // Generamos un array grande para enviar al Worker
      const numeros = Array.from({ length: 100000 }, (_, i) => i);
      workerLim.postMessage(numeros);
    });
  </script>
</body>
</html>
```

workerLimitaciones.js

```
onmessage = function(event) {
  // No podemos manipular el DOM directamente; solo podemos procesar datos.
  const datos = event.data; // Este es el array recibido
  const resultado = datos.map(num => num * 2); // Ejemplo de procesamiento
  // Buenas prácticas: intentar no devolver datos excesivamente grandes
  // si no es necesario, y coordinar la comunicación de forma eficiente.
  postMessage(resultado);
};
```

Se observa que aquí **no** accedemos a `document`, `window` o `alert()` en el Worker porque no está permitido. Del mismo modo, se requiere una comunicación vía mensajes para retornar la información procesada.

EJEMPLO 4: CÁLCULO ESTADÍSTICOS SIN BLOQUEO DE INTERFAZ

Imaginemos que tenemos una aplicación que, mientras el usuario puede hacer clic en botones, se encarga de **calcular estadísticas** de un gran volumen de datos en segundo plano. Este ejemplo muestra cómo delegar esa carga a un Worker sin bloquear la UI.

estadisticas.html

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8" />
  <title>Ejemplo Conjunto 1 - Estadísticas con Worker</title>
  <style>
    #contenedorBotones {
      margin-bottom: 10px;
    }
    #estadisticas {
      margin-top: 10px;
```

```
    background-color: #f5f5f5;
    padding: 10px;
  }
 </style>
</head>
<body>
 <h2>Estadísticas con Web Worker</h2>
 <div id="contenedorBotones">
  <button id="btnGenerarDatos">Generar Datos y Calcular Estadísticas</button>
  <button id="btnOtraAccion">Otra Acción (UI disponible)</button>
 </div>
 <div id="estado">En espera...</div>
 <div id="estadisticas"></div>

 <script>
  const btnGenerar = document.getElementById('btnGenerarDatos');
  const btnOtraAccion = document.getElementById('btnOtraAccion');
  const estado = document.getElementById('estado');
  const estadisticasDiv = document.getElementById('estadisticas');

  // Creamos un Worker para el cálculo de estadísticas
  const workerStats = new Worker('workerEstadisticas.js');

  // Cuando el Worker termine de procesar, actualizamos la interfaz
  workerStats.onmessage = (event) => {
   const { media, min, max } = event.data;
   estado.textContent = 'Cálculo completado.';
   estadisticasDiv.innerHTML = `
    <p>Media de la muestra: ${media.toFixed(2)}</p>
    <p>Valor mínimo: ${min}</p>
    <p>Valor máximo: ${max}</p>
   `;
  };

  workerStats.onerror = (event) => {
   estado.textContent = `Error en Worker: ${event.message}`;
```

```
    };

    // Botón para generar datos y pedirle al Worker que calcule
    btnGenerar.addEventListener('click', () => {
      estado.textContent = 'Generando datos y calculando...';
      estadisticasDiv.innerHTML = '';

      // Generamos un array con 1 millón de números aleatorios
      const datos = Array.from({ length: 1000000 }, () => Math.random() * 1000);

      // Enviamos el array al Worker
      workerStats.postMessage(datos);
    });

    // Botón para realizar otra acción "simulada" en la UI
    btnOtraAccion.addEventListener('click', () => {
      alert('La interfaz sigue respondiendo mientras se calculan estadísticas!');
    });
  </script>
</body>
</html>
```

workerEstadisticas.js

```
onmessage = function(event) {
  const datos = event.data; // Array numérico
  // Calculamos la media, el valor mínimo y el valor máximo
  // (Aquí podríamos realizar otros cálculos más complejos)
  let suma = 0;
  let min = Number.MAX_VALUE;
  let max = -Number.MAX_VALUE;

  for (let i = 0; i < datos.length; i++) {
    const valor = datos[i];
    suma += valor;
    if (valor < min) min = valor;
```

```
    if (valor > max) max = valor;
  }

  const media = suma / datos.length;
  // Enviamos los resultados al hilo principal
  postMessage({ media, min, max });
};
```

- La interfaz no se bloquea mientras se generan y calculan estadísticas de un array con un millón de números.
- El usuario puede hacer clic en “Otra Acción” para ver un `alert()`, demostrando que la UI permanece responsiva.

EJEMPLO 5: PROCESAMIENTO DE IMAGENES Y ASINCRONISMO

En este ejemplo, vamos a simular la carga de una imagen (o un archivo binario) y su posterior procesamiento con un Worker, mientras también realizamos una petición asíncrona con `fetch()` desde el hilo principal. Este ejercicio muestra cómo conviven la asincronía del Event Loop (peticiones de red) con la paralelización que ofrece el Worker.

procesarImagen.html

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8" />
  <title>Ejemplo Conjunto 2 - Procesamiento de Imagen</title>
  <style>
    #imagenPreview {
      max-width: 300px;
      margin: 10px 0;
    }
    #log {
      background: #f0f0f0;
```

```
    padding: 10px;
    margin-top: 10px;
  }
 </style>
</head>
<body>
 <h2>Procesamiento de Imagen con Web Worker</h2>
 <input type="file" id="inputFile" accept="image/*" />
 <button id="btnProcesar">Procesar Imagen</button>

 <div>
  <img id="imagenPreview" src="" alt="Vista previa" />
 </div>

 <div id="log"></div>

 <script>
  const inputFile = document.getElementById('inputFile');
  const btnProcesar = document.getElementById('btnProcesar');
  const imagenPreview = document.getElementById('imagenPreview');
  const logDiv = document.getElementById('log');
  let imagenSeleccionada = null;
  const workerImg = new Worker('workerImagen.js');
  // Mostrar la imagen seleccionada en pantalla
  inputFile.addEventListener('change', (event) => {
   const file = event.target.files[0];
   if (!file) return;
   const reader = new FileReader();
   reader.onload = function(e) {
    imagenPreview.src = e.target.result;
    imagenSeleccionada = e.target.result; // Base64 de la imagen
   };
   reader.readAsDataURL(file);
  });

  // Procesar imagen en el Worker
```

```
    btnProcesar.addEventListener('click', () => {
      if (!imagenSeleccionada) {
        alert('Selecciona primero una imagen');
        return;
      }

      log('Enviando imagen al Worker para procesar...');
      // Enviamos la imagen codificada en Base64 al Worker
      workerImg.postMessage(imagenSeleccionada);
    });
    // Respuesta del Worker
    workerImg.onmessage = (event) => {
      // Recibimos la imagen procesada o un mensaje de estado
      if (event.data.tipo === 'log') {
        log(event.data.mensaje);
      } else if (event.data.tipo === 'resultado') {
        log('Imagen procesada con éxito. Puedes visualizar el resultado en consola o aplicarlo a un canvas.');
      }
    };
    // Ejemplo de asincronía adicional: una petición fetch que ocurre en el hilo principal
    // (Se ejecuta en paralelo a las operaciones del Worker)
    fetch('https://jsonplaceholder.typicode.com/posts/1')
      .then(response => response.json())
      .then(data => {
        log(`Petición fetch completada: Título del post: "${data.title}"`);
      })
      .catch(error => {
        log(`Error en fetch: ${error}`);
      });
    // Función para registrar mensajes en pantalla
    function log(mensaje) {
      logDiv.innerHTML += `<p>${mensaje}</p>`;
    }
  </script>
</body>
</html>
```

workerImagen.js

```
// Simulamos un procesamiento de imagen (por ejemplo, aplicando un filtro simple)
onmessage = function(event) {
  const imagenBase64 = event.data;
  // En un caso real, podríamos convertir la imagen a un ArrayBuffer,
  // aplicar filtros pixel a pixel y volver a convertir a Base64, etc.
  // Por simplicidad, simulamos un tiempo de procesamiento y enviamos logs intermedios
  postMessage({ tipo: 'log', mensaje: 'Iniciando procesamiento de imagen...' });

  // Simulamos un retardo (por ejemplo, 2 segundos)
  setTimeout(() => {
    postMessage({ tipo: 'log', mensaje: 'Procesamiento intermedio...' });
  }, 1000);
  setTimeout(() => {
    // Al finalizar, devolvemos el resultado (aquí, por ejemplo, podríamos devolver la imagen procesada)
    postMessage({ tipo: 'resultado', data: imagenBase64 });
  }, 3000);
};
```

- Este ejemplo combina la lectura de archivos (`FileReader`) y la transformación simulada en un Worker, junto con una petición `fetch` en el hilo principal.
- Mientras el Worker está ocupado, la petición `fetch` se realiza de forma asíncrona sin bloqueo.

10.7. CONCLUSIÓN

Los Web Workers representan uno de los avances más significativos para el desarrollo web moderno al permitir que JavaScript ejecute tareas intensivas de CPU sin bloquear la experiencia de usuario. Gracias a ellos, ahora es posible aprovechar múltiples hilos de ejecución y hacer un uso más eficaz de los recursos del sistema. No obstante, su uso no está exento de limitaciones, principalmente la imposibilidad de manipular el DOM directamente desde un Worker y la necesidad de comunicar información mediante mensajes.

En todo caso, cuando el volumen de datos a procesar sea alto, es buena idea recurri a los web workers. El diseño cuidadoso del intercambio de mensajes y la definición de responsabilidades claras entre el hilo principal y los Workers son claves para crear aplicaciones web avanzadas, de gran rendimiento y libres de bloqueos. A la hora de mostrar o actualizar datos en la interfaz, confía la responsabilidad al hilo principal. Delega a los Workers aquellas tareas que exijan una gran cantidad de recursos, como cálculos matemáticos complejos, procesado de ficheros o la preparación de grandes volúmenes de datos. De este modo, tu aplicación se beneficiará de una interacción ágil y una experiencia de usuario mucho más fluida.