ASINCRONISMO EN JAVASCRIPT

TEMA 05 – APIs NATIVAS PARA ASINCRONISMO

5.1. Introducción

Existen varias formas de trabajar con asincronismo en JavaScript:

• **Callbacks** (la forma más antigua y básica).

• **Web APIs** y librerías del navegador que nos permiten programar comportamientos asíncronos, principalmente:

- setTimeout y setInterval.
- fetch.
- Xmlhttprequest.
- Promises.
- async/await.

En este documento, nos centraremos en describir setTimeout y setInterval como ejemplos de temporizadores de JavaScript, así como en introducir la función fetch para la realización de solicitudes HTTP (peticiones a un servidor remoto para obtener o enviar datos)

5.1. Uso de setTimeout y setInterval

¿Qué son?

• setTimeout: Programa la ejecución de un código (función o fragmento de código) transcurrido un período de tiempo que se especifica en milisegundos. Es decir, se utiliza para retrasar la ejecución de algo.

• setInterval: Programa la ejecución periódica de un código, de modo que se ejecuta una vez transcurrido el período de tiempo especificado y se repite de forma indefinida (o hasta que lo detengamos).

La clave para entender por qué se consideran APIs asíncronas es que, cuando se programa una tarea con setTimeout o setInterval, JavaScript no se queda “esperando” a que pase el tiempo indicado, sino que registra el evento en la cola del **event loop**. Mientras, el intérprete sigue ejecutando el resto del script. Una vez que se cumpla el plazo en milisegundos establecido, el motor de JavaScript recuperará la función o código asociado y lo ejecutará.

Cómo se relacionan setTimeout y setInterval con el event loop y la task queue

setTimeout

Cuando usas setTimeout(callback, tiempo):

1. Se registra un temporizador y se indica que, **una vez transcurridos** los milisegundos especificados (tiempo), se añada la función callback a la cola de tareas.

2. **Importante**: el temporizador no “garantiza” que el callback se ejecute exactamente al milisegundo exacto. Solo se asegura de que, como mínimo, pasen esos milisegundos antes de que el callback ingrese a la cola de tareas.

3. Pasado el tiempo configurado, se añade la tarea (el callback) a la **task queue**.

4. El **event loop**, cuando termine de ejecutar la pila principal, atenderá esta nueva tarea. Si en ese momento no hay nada de mayor prioridad, tomará la tarea de la cola y la ejecutará.

Ejemplo

```
console.log("Inicio");
setTimeout(() => {
  console.log("Mensaje retrasado");
}, 2000);
console.log("Fin");
```

• console.log("Inicio") y console.log("Fin") se ejecutan de inmediato (sincrónicamente).

• El setTimeout se configura para 2 segundos. Cuando estos transcurran, el callback (() => { console.log("Mensaje retrasado") }) se envía a la **task queue**.

• El event loop lo ejecutará en cuanto finalicen todas las tareas sincrónicas que se encuentren en la cola o en el stack.

setInterval

setInterval(callback, tiempo) funciona de forma similar a setTimeout, pero con la diferencia de que, al cumplirse el intervalo de tiempo, **la tarea (callback) se vuelve a programar** para ser ejecutada de nuevo al cabo de otros tantos milisegundos. Esto se repite indefinidamente o hasta que sea cancelado con clearInterval. El proceso es:

1. Se configura un intervalo repetitivo cada tiempo milisegundos.

2. Cada vez que transcurre ese lapso, el **motor de JavaScript** añade el callback a la cola de tareas.

3. El **event loop** extrae el callback de la cola y lo ejecuta cuando el call stack está libre.

Ejemplo

```
let contador = 0;
const idIntervalo = setInterval(() => {
  contador++;
  console.log(`Contador: ${contador}`);
  if (contador >= 5) {
    clearInterval(idIntervalo);
    console.log("Intervalo detenido.");
  }
}, 1000);
```

• Cada 1000 milisegundos, la tarea se añade a la cola de tareas.

• El event loop la atiende cuando puede (normalmente de inmediato si no hay otra tarea anterior).

• Se incrementa el contador y se muestra en consola.

• Al llegar a 5, se cancela el intervalo.

Diferencias y consideraciones

1. **No bloquean el hilo**: Ni setTimeout ni setInterval bloquean la ejecución del resto de código. Crean “tareas pendientes” que se ponen en la cola de tareas. El motor de JavaScript puede seguir ejecutando otra lógica mientras espera a que se cumpla el tiempo o el intervalo.

2. **Orden de ejecución**: Las tareas asíncronas solo se ejecutan cuando el **call stack está vacío**. Esto implica que, si el programa principal está haciendo un cálculo muy pesado, puede haber un retraso adicional.

3. **Precisión**: El tiempo de retraso es orientativo. No está garantizado que se cumpla exactamente, ya que depende de la carga que el hilo principal tenga en ese momento, aunque de forma habitual, la precisión es razonable.

4. **Identificador de intervalo**: Tanto setInterval como setTimeout devuelven un identificador (un número entero) que sirve para detener la ejecución programada. Para setTimeout, se usa clearTimeout(), y para setInterval, se usa clearInterval().

Tanto setTimeout como setInterval programan ejecuciones asíncronas controladas por el event loop, utilizando la task queue como lugar de espera antes de su ejecución. Esta arquitectura permite que JavaScript siga ejecutando el resto del programa mientras esperan el tiempo establecido o mientras se planifican repeticiones periódicas, ofreciendo una experiencia fluida y no bloqueante.

5.2. API fetch para solicitudes HTTP

¿Qué es fetch?

La API fetch es una de las Web APIs más comunes para realizar solicitudes HTTP desde JavaScript. En versiones antiguas de JavaScript, se usaba XMLHttpRequest (XHR) para realizar peticiones AJAX, pero fetch ha ido ganando terreno gracias a su diseño basado en Promesas, que facilita la lectura y escritura del código.

Sintaxis general:

```
fetch(url, [opciones])
  .then(respuesta => {
    // Procesar la respuesta
  })
  .catch(error => {
    // Manejar el error
  });
```

Donde:

• url es la dirección del recurso que se desea obtener o la ruta a la que se quiere enviar datos.

• opciones es un objeto que permite configurar la petición (método HTTP como GET, POST, PUT, etc., cabeceras, cuerpo de la petición, entre otros).

Ejemplo1: fetch

Si quisieramos realizar una solicitud de datos a un servicio externo que nos devuelva, por ejemplo, un listado de usuarios en formato JSON, podríamos hacer algo como lo siguiente:

```
fetch("https://jsonplaceholder.typicode.com/users")
  .then(response => {
    // Comprobamos si la respuesta es correcta (código HTTP 200-299)
    if (!response.ok) {
```

```
      throw new Error(`Error en la petición: ${response.status}`);
    }
    // Convertimos la respuesta en formato JSON
    return response.json();
  })
  .then(data => {
    console.log("Listado de usuarios:", data);
  })
  .catch(error => {
    console.error("Hubo un problema con la petición:", error);
  });
```

Flujo de ejecución:

1. Se llama a fetch pasandole la URL del recurso a consumir.

2. A través del objeto response se realizan dos acciones, la primera, si ha habido éxito al realizar la petición y, la segunda, el “parseo” de los datos obtenidos en formato JSON al formato nativo de JavaScript. Mientras se realiza la petición, el programa no se bloquea y continúa su ejecución.

3. Los datos ya en formato JavaScript se obtienen en el objeto data y se muestran por consola.

4. Si se produce un error (e.g., un problema de conexión o un código HTTP fuera de rango 2xx), se captura en el bloque catch.

Ejemplo 2: setTimeout + fetch

Este ejemplo realiza una solicitud HTTP para obtener datos de un servidor, pero antes de hacerlo esperamos un tiempo determinado, simulando un “retraso controlado” antes de disparar la petición.

```
console.log("Programa iniciado. Preparando para hacer fetch en 3 segundos...");

// Esperamos 3 segundos antes de hacer la petición
setTimeout(function() {
```

```
  console.log("Iniciando fetch...");
  fetch("https://jsonplaceholder.typicode.com/posts/1")
    .then(function(response) {
      if (!response.ok) {
        throw new Error(`Error en la solicitud: ${response.status}
`);
      }
      return response.json();
    })
    .then(function(data) {
      console.log("Datos recibidos (tras 3 segundos de retraso):",
data);
    })
    .catch(function(error) {
      console.error("Hubo un error en la petición:", error);
    });
}, 3000);
```

Explicación:

1. Se imprime un mensaje de inicio y se programa una función con setTimeout que se disparará a los 3 segundos.

2. Pasados esos 3 segundos, se ejecuta la función interna, que realiza un fetch a jsonplaceholder.typicode.com para obtener el detalle de un “post” con ID=1.

3. Una vez que los datos llegan, se muestran en la consola.

Ejemplo 3: setInterval + fetch

Aquí realizamos una consulta al servidor cada X segundos para actualizar cierta información, simulando un sondeo (polling). En este caso, cada 5 segundos se obtiene un recurso y se muestra por consola. Si la respuesta indica algún error, lo manejaremos. También añadimos la posibilidad de detener el sondeo con un botón en la página.

HTML:

```html
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Ejemplo setInterval + fetch</title>
</head>
<body>
  <button id="btnDetener">Detener sondeo</button>
  <script src="app.js"></script>
</body>
</html>
```

JavaScript:

```javascript
const intervaloSondeo = setInterval(function() {
  console.log("Realizando petición al servidor...");
  fetch("https://jsonplaceholder.typicode.com/todos/1")
    .then(function(response) {
      if (!response.ok) {
        throw new Error(`Error en la solicitud: ${response.status}
`);
      }
      return response.json();
    })
    .then(function(data) {
      console.log("Datos obtenidos:", data);
    })
    .catch(function(error) {
      console.error("Error al obtener los datos:", error);
    });
}, 5000);
// Escuchamos el evento de clic en el botón para detener el sondeo
const btnDetener = document.getElementById("btnDetener");
```

```
btnDetener.addEventListener("click", function() {
  clearInterval(intervaloSondeo);
  console.log("Sondeo detenido.");
});
```

Conclusión

Como puedes observar, cada una de estas APIs se puede usar de forma individual, pero también se combinan de manera frecuente en proyectos reales para lograr comportamientos más complejos:

- setTimeout y setInterval se encargan de temporizar acciones, permitiendo programar ejecuciones retrasadas o periódicas.
- fetch es clave para realizar peticiones HTTP y manejar datos remotos sin bloquear el hilo principal.

Al unirlos, JavaScript adquiere la capacidad de realizar sondeos recurrentes a un servidor, programar acciones puntuales tras cierto tiempo o mostrar datos tras un periodo controlado de espera. Estos mecanismos asíncronos son fundamentales en el desarrollo moderno de aplicaciones web, pues permiten ofrecer experiencias dinámicas y reactivas a los usuarios.