# ASINCRONISMO EN JAVASCRIPT

## TEMA 06 – XHR Y AJAX

## 6.1. Introducción

A lo largo de la historia de JavaScript, uno de los elementos clave que ha revolucionado la comunicación asíncrona entre el cliente y el servidor es el concepto de **AJAX** (Asynchronous JavaScript and XML). AJAX permite actualizar partes de una página web sin recargarla completamente, lo que mejora significativamente la experiencia del usuario. En este contexto, la interfaz **XMLHttpRequest (XHR)**, ha sido una herramienta esencial para implementar esta funcionalidad. Aunque en la actualidad se emplean métodos más modernos como **fetch()**, XMLHttpRequest sigue siendo un componente fundamental para comprender los principios básicos del intercambio de datos en segundo plano, sin bloquear la interacción del usuario con la página web.

En este apartado, exploraremos qué es XMLHttpRequest, cómo se utiliza para realizar solicitudes HTTP (como GET o POST) y procesar datos en formato JSON o XML. Además, presentaremos un ejemplo práctico que ilustra su funcionamiento y, finalmente, compararemos brevemente sus características con las de la función fetch para destacar sus principales diferencias y ventajas.

### 6.1. Qué es XMLHttpRequest y cómo se usa para solicitudes HTTP

**XMLHttpRequest** (a menudo abreviado como XHR) es un objeto proporcionado por los navegadores web que permite realizar solicitudes HTTP y HTTPS de forma asíncrona. Su principal ventaja es la de habilitar la comunicación con el servidor sin necesidad de recargar toda la página, lo que se conoce comúnmente como **AJAX** (Asynchronous JavaScript and XML). Con XMLHttpRequest, podemos:

1. **Enviar y recibir datos en segundo plano** mientras el usuario sigue interactuando con la aplicación.

2. **Hacer peticiones a diferentes métodos HTTP** (GET, POST, PUT, DELETE, etc.).

3. **Trabajar con distintos formatos de datos**: JSON, XML, texto plano, HTML, etc.

4. **Personalizar cabeceras (headers)** en nuestras solicitudes y, de igual modo, leer las cabeceras de respuesta.

Antes de la llegada de fetch(), XMLHttpRequest era la forma estándar de realizar llamadas AJAX. Aun con la aparición de técnicas más modernas, es importante entenderlo en detalle ya que sigue siendo ampliamente utilizado en proyectos ya implantados.

### Creación de una instancia de XMLHttpRequest

El primer paso para utilizar esta interfaz es crear una instancia del objeto:

```
let xhr = new XMLHttpRequest();
```

Una vez creada la instancia xhr, podemos usarla para preparar y enviar solicitudes a un servidor.

### Solicitud con XMLHttpRequest

1. **Definir el tipo de solicitud y la URL**: se utiliza el método open() para indicar el método HTTP (GET, POST, etc.) y la dirección (URL) a la que vamos a enviar la petición.

```
xhr.open('GET', 'ruta/del/servidor', true);
```

El tercer parámetro indica si la petición será asíncrona (true) o síncrona (false). Prácticamente siempre desearemos trabajar de forma asíncrona.

2. **Configurar cabeceras y eventos** (opcional):

• Podemos ajustar cabeceras con setRequestHeader().

• Definir funciones callback o manejadores de eventos para controlar lo que sucederá cuando la respuesta llegue o en caso de error.

•Por ejemplo, podemos asignar una función al evento onreadystatechange o utilizar xhr.addEventListener('readystatechange', callback).

3. **Enviar la solicitud**: mediante el método send() se lanza la petición al servidor. Si utilizamos el método GET, normalmente no se envía cuerpo de la petición; en cambio, para POST se suele enviar un cuerpo (datos del formulario, JSON, etc.).

```
xhr.send();
```

4. **Recibir y procesar la respuesta**: cuando el servidor responde, podemos obtener su contenido a través de propiedades como xhr.responseText o xhr.responseXML, dependiendo del formato con que se haya devuelto la información.

El evento clave para detectar el momento en que la respuesta está lista es readystatechange. Dentro de este evento, comprobamos si xhr.readyState === 4 y xhr.status === 200, lo que indica que la respuesta se ha recibido con éxito.

• readyState puede tener los valores:

- 0: no se ha inicializado (se acaba de crear el objeto).
- 1: conexión establecida con open().
- 2: se han recibido las cabeceras de la respuesta.
- 3: está recibiendo el cuerpo de la respuesta.
- 4: la respuesta ha sido recibida por completo.

• status típicamente es 200 si la solicitud se ha realizado con éxito, y variará en función de otros casos (404, 500, 401, etc.).

**Ejemplo: Solicitud GET y manejo de datos JSON**

Veamos un ejemplo práctico de cómo realizar una **solicitud GET** con XMLHttpRequest y procesar la respuesta, tanto en formato **JSON** como en **XML**. Este ejemplo puede servir como una plantilla, la cual posteriormente se

podrá adaptar a otros métodos como POST, PUT o DELETE, si fuera necesario.

Supongamos que en el servidor tenemos un endpoint que devuelve información en formato JSON, por ejemplo, en la ruta https://api.ejemplo.com/datos. El código para realizar la solicitud y manejar la respuesta JSON podría ser el siguiente:

```
// Creamos la instancia
let xhr = new XMLHttpRequest();
// Abrimos la solicitud indicando método y URL
xhr.open('GET', 'https://api.ejemplo.com/datos', true);
// Opcional: podemos establecer cabeceras si es necesario
// xhr.setRequestHeader('Content-Type', 'application/json');

// Definimos la función que se ejecutará cada vez que cambie el estado de la petición
xhr.onreadystatechange = function () {
  // Verificamos que haya finalizado la recepción de la respuesta
  if (xhr.readyState === 4) {
    // Comprobamos si la respuesta se recibió correctamente
    if (xhr.status === 200) {
      // Convertimos el texto de respuesta a un objeto JSON
      let datos = JSON.parse(xhr.responseText);
    // Aquí podemos manipular los datos según nuestras necesidades
      console.log('Datos recibidos (JSON):', datos);
    } else {
      // Manejo de error
        console.error('Error al realizar la solicitud. Código de estado:', xhr.status);
    }
  }
};
```

```
// Enviamos la solicitud
xhr.send();
```

**Explicación:**

1. **Creación**: new XMLHttpRequest().

2. **Configuración**: open('GET', 'https://api.ejemplo.com/datos', true) abre una petición GET a la URL especificada con modo asíncrono.

3. **Control del evento**: la función anónima asignada a onreadystatechange se ejecuta cada vez que el estado de la petición cambia.

4. **Validación de respuesta**: comprobamos si readyState === 4 (respuesta completa) y status === 200 (éxito).

5. **Procesamiento de datos**: empleamos JSON.parse para convertir el texto en un objeto JSON.

**Ejemplo: Solicitud GET y manejo de datos XML**

Si el servidor nos devuelve un documento en formato XML, podemos accederlo a través de la propiedad responseXML. Veamos un ejemplo:

```
let xhr = new XMLHttpRequest();
xhr.open('GET', 'https://api.ejemplo.com/datos.xml', true);
xhr.onreadystatechange = function () {
  if (xhr.readyState === 4) {
    if (xhr.status === 200) {
      let xmlDoc = xhr.responseXML;
      if (xmlDoc) {
        // Por ejemplo, si el nodo raíz se llama <raiz>
        let raiz = xmlDoc.getElementsByTagName('raiz')[0];
        console.log('Nodo raíz:', raiz.nodeName);
       // Podemos seguir explorando el DOM XML según sea necesario
      } else {
          console.error('No se pudo interpretar la respuesta como
XML.');
```

```
      }
    } else {
        console.error('Error al realizar la solicitud. Código de estado:', xhr.status);
    }
  }
};

xhr.send();
```

**Explicación:**

1. Una vez completada la petición, usamos xhr.responseXML para obtener un documento DOM que representa el archivo XML.

2. Podemos navegar el DOM resultante para extraer la información deseada mediante métodos como getElementsByTagName, getAttribute, etc.

3. En caso de que la respuesta no fuera realmente XML o hubiera algún problema, responseXML podría devolverse como null.

### 6.3. Comparación con fetch

**XMLHttpRequest** fue la tecnología pionera para realizar llamadas asíncronas en JavaScript. Sin embargo, a lo largo del tiempo apareció el método **fetch()**, que ofrece una **API más moderna** y basada en **promesas**, haciéndola más legible y permitiendo un uso más natural de funcionalidades recientes como async/await. He aquí una comparación:

1. **Sintaxis y estilo de programación**

• XMLHttpRequest: utiliza eventos (onreadystatechange) y callbacks para controlar la evolución de la solicitud. Esto puede llevar a estructuras de código algo más complejas, especialmente si se anidan varias peticiones.

• fetch: implementa promesas, por lo que facilita la composición de peticiones encadenadas, el manejo de errores con catch() y la legibilidad del código con la sintaxis async/await.

2. **Soporte y compatibilidad**

• Ambas son ampliamente soportadas por los navegadores modernos.

• XMLHttpRequest existe desde antes de fetch, lo que lo hace compatible con la gran mayoría de navegadores, incluyendo versiones más antiguas.

3. **Manejo de la respuesta**

• XMLHttpRequest: se accede mediante propiedades como responseText o responseXML.

• fetch: la respuesta se gestiona a través de métodos asíncronos como .json() o .text(), entre otros.

4. **Eventos y errores**

• XMLHttpRequest: propone distintos eventos (onload, onerror, onreadystatechange) para manejo de éxito, error y progreso.

• fetch: trabaja con promesas; el estado de la petición se maneja con .then() y .catch(). Los errores de red se capturan fácilmente con .catch(). Para manejar errores de estado (como 404, 500, etc.), se verifica manualmente la propiedad response.ok.

5. **Progresos de carga y otros detalles**

• XMLHttpRequest: resulta más sencillo monitorear el progreso de la carga con el evento onprogress.

• fetch: todavía no cuenta con un soporte nativo tan directo para seguimiento del progreso en la descarga del cuerpo de la respuesta (existen métodos basados en ReadableStream, pero la implementación y manipulación es más compleja).

## 6.4. Ejemplos

### Ejemplo 1

En este primer fragmento de código se demuestra lo esencial: crear una instancia de **XMLHttpRequest**, abrir una conexión GET a una URL y capturar el resultado.

```
// EJEMPLO 1: Uso básico de XMLHttpRequest para una solicitud GET (texto plano o HTML)
function cargarContenidoTexto() {
  // 1. Crear instancia de XMLHttpRequest
  let xhr = new XMLHttpRequest();
  // 2. Configurar la petición (método y URL)
  xhr.open('GET', 'https://ejemplo.com/contenido.txt', true);
  // 3. Manejadores de evento: onreadystatechange o onload
  xhr.onreadystatechange = function () {
    // Verificar si la respuesta ha llegado (readyState = 4) y es correcta (status = 200)
    if (xhr.readyState === 4 && xhr.status === 200) {
      // 4. Obtener el texto de la respuesta
      let respuesta = xhr.responseText;
     // 5. Mostrar la respuesta en consola o manipularla en el DOM
      console.log('Contenido recibido:', respuesta);
    }
  };
  // 6. Enviar la solicitud
  xhr.send();
}
// Llamada a la función
cargarContenidoTexto();
```

### Explicación

• Creación de la instancia con new XMLHttpRequest().

• Uso de open('GET', 'url', true) para una petición asíncrona.

• Validación de readyState === 4 y status === 200 para asegurarnos de que la respuesta sea exitosa.

• Obtención de la respuesta con xhr.responseText (útil cuando el servidor devuelve texto plano o HTML).

**Ejemplo 2**

Este segundo ejemplo muestra cómo recuperar datos JSON desde un endpoint y convertirlos a un objeto JavaScript manipulable.

```
// EJEMPLO 2: Solicitud GET y manejo de JSON
function cargarDatosJSON() {
  // 1. Crear la instancia
  let xhr = new XMLHttpRequest();
  // 2. Definir la URL que devuelve datos en formato JSON
  xhr.open('GET', 'https://api.ejemplo.com/datos', true);
  // 3. Manejador de eventos
  xhr.onreadystatechange = function () {
    if (xhr.readyState === 4) {
      if (xhr.status === 200) {
        // 4. Transformar el texto de la respuesta en un objeto
JavaScript
        let datos = JSON.parse(xhr.responseText);
        // 5. Manipular los datos (ejemplo: mostrarlos en consola)
        console.log('Datos JSON recibidos:', datos);
      } else {
        // Manejo de error si la respuesta no es 200
        console.error('Error al cargar datos JSON. Código de
estado:', xhr.status);
      }
    }
  };
  // 6. Enviar la solicitud
  xhr.send();
```

```
}
// Llamada a la función
cargarDatosJSON();
```

**Explicación**

• El uso de JSON.parse(xhr.responseText) para convertir de texto plano a objeto JavaScript.

• El control de errores en caso de no obtener un código de estado 200.

**Ejemplo 3**

Este fragmento de código muestra cómo se trabajan datos en formato XML y se accede a su contenido como un DOM XML.

```
// EJEMPLO 3: Solicitud GET y manejo de XML
function cargarDatosXML() {
  // 1. Crear la instancia
  let xhr = new XMLHttpRequest();
  // 2. Definir la URL que devuelve un documento XML
  xhr.open('GET', 'https://api.ejemplo.com/datos.xml', true);
  // 3. Evento para manejar la respuesta
  xhr.onreadystatechange = function () {
    if (xhr.readyState === 4) {
      if (xhr.status === 200) {
        // 4. Recibimos el DOM XML
        let xmlDoc = xhr.responseXML;
        // 5. Manipular el DOM XML (ejemplo: leer etiquetas
"usuario")
        if (xmlDoc) {
          let usuarios = xmlDoc.getElementsByTagName('usuario');
          for (let i = 0; i < usuarios.length; i++) {
            console.log('Usuario #' + (i + 1) + ':',
usuarios[i].textContent);
          }
```

```
        } else {
          console.error('No se pudo interpretar la respuesta como XML.');
        }
      } else {
        console.error('Error al cargar datos XML. Código de estado:', xhr.status);
      }
    }
  };
  // 6. Enviar la solicitud
  xhr.send();
}
// Llamada a la función
cargarDatosXML();
```

**Explicación**

- El uso de xhr.responseXML para obtener un objeto DOM.
- La posterior manipulación con métodos del DOM (por ejemplo, getElementsByTagName).

**Ejemplo 4**

En este ejemplo, veremos cómo sería la **misma** solicitud GET implementada con **fetch()** en comparación a **XMLHttpRequest**, para que el alumnado entienda las diferencias sintácticas y de manejo de promesas.

```
// EJEMPLO 4A: Petición GET con XMLHttpRequest
function cargarConXHR() {
  let xhr = new XMLHttpRequest();
  xhr.open('GET', 'https://api.ejemplo.com/datos', true);
  xhr.onreadystatechange = function () {
    if (xhr.readyState === 4 && xhr.status === 200) {
      console.log('XHR:', JSON.parse(xhr.responseText));
```

```
    }
  };
  xhr.send();
}
// EJEMPLO 4B: Petición GET con fetch
async function cargarConFetch() {
  try {
    let respuesta = await fetch('https://api.ejemplo.com/datos');
    if (respuesta.ok) {
      let datos = await respuesta.json();
      console.log('fetch:', datos);
    } else {
      console.error('Error con fetch. Código de estado:',
respuesta.status);
    }
  } catch (error) {
    console.error('Error de red u otro problema:', error);
  }
}
// Llamadas a las funciones
cargarConXHR();
cargarConFetch();
```

**Diferencias clave**

1. **XHR** usa readyState y callbacks (en este caso onreadystatechange) para detectar cuándo la respuesta está lista.

2. **fetch** se basa en promesas, lo que permite una sintaxis más limpia con async/await.

3. En el caso de fetch, debemos manejar el error de estado manualmente verificando respuesta.ok.

**Ejemplo 5**

Para completar la visión de cómo funciona XMLHttpRequest, aquí se muestra un ejemplo de envío de datos (una solicitud POST) con un objeto JSON:

```
// EJEMPLO 5: Petición POST con envío de JSON
function enviarDatosJSON() {
  let xhr = new XMLHttpRequest();
  xhr.open('POST', 'https://api.ejemplo.com/guardar', true);
  // Cabecera para indicar que enviamos JSON
  xhr.setRequestHeader('Content-Type', 'application/json;charset=UTF-8');
  // Manejar la respuesta
  xhr.onreadystatechange = function () {
    if (xhr.readyState === 4) {
      if (xhr.status === 200) {
        console.log('Datos enviados correctamente:', xhr.responseText);
      } else {
        console.error('Error al enviar datos. Código de estado:', xhr.status);
      }
    }
  };
  // Objeto JavaScript con la información que vamos a enviar
  let data = {
    nombre: 'Juan',
    edad: 30,
    ocupacion: 'Desarrollador',
  };
  // Convertimos a JSON y lo enviamos
  xhr.send(JSON.stringify(data));
}
// Llamada a la función
enviarDatosJSON();
```

**Explicación:**

• Uso de POST y setRequestHeader para indicar el tipo de contenido.

• JSON.stringify para transformar el objeto JavaScript a JSON antes de enviarlo.

**Ejemplo 6: Obtener un recurso en JSON y, a continuación, enviar un formulario por POST (XHR)**

1. Primero hacemos una petición GET para obtener datos en JSON y mostramos esa información.

2. Luego realizamos una **petición POST** para enviar un formulario (también en JSON), aprovechando que el usuario podría haber rellenado algún campo y se quiere guardar en el servidor.

```
function flujoCompletoXHR() {
  // 1. Petición GET para datos JSON
  let xhrGet = new XMLHttpRequest();
     xhrGet.open('GET',   'https://api.ejemplo.com/datosIniciales',
true);
  xhrGet.onreadystatechange = function () {
    if (xhrGet.readyState === 4) {
      if (xhrGet.status === 200) {
        let datosRecibidos = JSON.parse(xhrGet.responseText);
        console.log('Datos iniciales:', datosRecibidos);
// 2. Simular un formulario que completamos y enviamos por POST
        let formulario = {
          nombre: 'María',
          curso: 'Desarrollo Web',
        };
        // 3. Crear nueva instancia para la petición POST
        let xhrPost = new XMLHttpRequest();
                                                xhrPost.open('POST',
'https://api.ejemplo.com/guardarFormulario', true);
```

```
                        xhrPost.setRequestHeader('Content-Type',
'application/json;charset=UTF-8');
        xhrPost.onreadystatechange = function () {
          if (xhrPost.readyState === 4) {
            if (xhrPost.status === 200) {
                  console.log('Formulario guardado correctamente:',
xhrPost.responseText);
            } else {
                console.error('Error al guardar formulario. Código
de estado:', xhrPost.status);
            }
          }
        };
        // 4. Enviamos el formulario
        xhrPost.send(JSON.stringify(formulario));
      } else {
         console.error('Error al obtener datos iniciales. Código de
estado:', xhrGet.status);
      }
    }
  };
  xhrGet.send();
}
// Invocamos la función
flujoCompletoXHR();
```

En este **flujo** se muestra el manejo secuencial de peticiones: primero un **GET** y luego un **POST**, todo ello usando XMLHttpRequest. Se encadenan las operaciones en el onreadystatechange de la primera llamada, asegurándonos de no enviar el POST hasta que la respuesta GET haya sido satisfactoria.

**Ejemplo 7: Comparar en una sola función un proceso con XHR y otro con fetch**

Este último ejemplo integra lo siguiente:

1. Realiza una petición **GET** con **XMLHttpRequest** para obtener datos en JSON.

2. Realiza una petición **GET** con **fetch()** para otra URL distinta, comparando la forma de capturar y procesar esos datos.

```
async function procesoMixtoXHRyFetch() {
  // --- Parte A: Obtener datos con XMLHttpRequest ---
  const xhr = new XMLHttpRequest();
  xhr.open('GET', 'https://api.ejemplo.com/datosXHR', true);
  xhr.onreadystatechange = function () {
    if (xhr.readyState === 4 && xhr.status === 200) {
      let datosXHR = JSON.parse(xhr.responseText);
      console.log('Datos obtenidos con XHR:', datosXHR);
    }
  };
  xhr.send();
  // --- Parte B: Obtener datos con fetch ---
  try {
    let respuesta = await fetch('https://api.ejemplo.com/datosFetch');
    if (respuesta.ok) {
      let datosFetch = await respuesta.json();
      console.log('Datos obtenidos con fetch:', datosFetch);
    } else {
      console.error('Error fetch. Código de estado:', respuesta.status);
    }
  } catch (error) {
    console.error('Error de red u otro problema con fetch:', error);
  }
}
// Llamada a la función para ejecutar ambos procesos
procesoMixtoXHRyFetch();
```

En este caso, tanto la parte XHR como la parte fetch se ejecutan de forma independiente dentro de una misma función, ilustrando la coexistencia de ambas soluciones. Podríamos incluso usar los resultados de la primera (XHR) para condicionar el fetch, pero en este ejemplo se muestran como peticiones paralelas que no dependen una de la otra.

### 6.5. Conclusión

Si bien fetch es la opción recomendada para la mayoría de proyectos nuevos por su estilo asíncrono más moderno y limpio, entender a fondo XMLHttpRequest es importante para mantener proyectos antiguos (legacy), asegurarse compatibilidad con versiones de navegadores más viejas y comprender los fundamentos históricos de la comunicación asíncrona en JavaScript.

En resumen, **XMLHttpRequest** sigue siendo una pieza clave del ecosistema JS. A lo largo de los años, la comunidad ha ido evolucionando hacia nuevas interfaces como fetch, pero **XHR** conserva su relevancia tanto en términos de compatibilidad como para escenarios donde se necesite un control más bajo nivel de ciertos aspectos (por ejemplo, seguimiento de progreso con eventos). Conocer ambas opciones permite elegir la mejor herramienta para cada situación y mantener o modernizar proyectos según sea necesario.