Capítulo 3

Comunicación Cliente-Servidor en Aplicaciones Web

3.1. Fundamentos

Las aplicaciones web se basan en el Protocolo de Transferencia de Hipertexto (HTTP), un modelo de comunicación sin estado (stateless). En este modelo, el cliente (navegador) inicia una solicitud (**Request**), y el servidor responde con los recursos solicitados (**Response**). Históricamente, cualquier nueva solicitud de datos o interacción del usuario requería una navegación a una página nueva o la recarga completa de la página actual.

Decir que HTTP es un protocolo sin estado significa que cada petición que un cliente hace a un servidor es independiente de las demás. El servidor no guarda información sobre las solicitudes anteriores: trata cada nueva petición como si fuera la primera vez que ese cliente se comunica con él.

La evolución de las aplicaciones web demandó una forma de actualizar partes específicas del Document Object Model (DOM) sin recargar la página. El uso de la **comunicación asíncrona**, conocida genéricamente como **AJAX**, resolvió este problema al permitir que **JavaScript intercambie datos con el servidor en segundo plano, actualizando solo los componentes necesarios y simulando así una coherencia de estado en el cliente**.

Figura 3.1: Request - Response

3.2. AJAX

Figura 3.2: Ajax

AJAX, acrónimo de **Asynchronous JavaScript and XML**, define un **conjunto de tecnologías** que permiten a las aplicaciones web intercambiar datos con servidores de manera asíncrona.

Aunque el XML es parte de su nombre, el estándar de intercambio de datos dominante hoy en día es JSON. La primera tecnología que hizo esto posible en el navegador fue la interfaz **XMLHttpRequest**.

Las tecnologías que conforman ajax son:

- **HTML** para el lenguaje principal y **CSS** para la presentación.
- El Modelo de objetos del documento (**DOM**) para datos de visualización dinámicos y su interacción.
- **XML** para el intercambio de datos. Cómo ya se ha comentado, hoy en día sustituido por **JSON**.
- El objeto **XMLHttpRequest** para la comunicación asíncrona.
- El lenguaje de programación **JavaScript** para unir todas estas tecnologías.

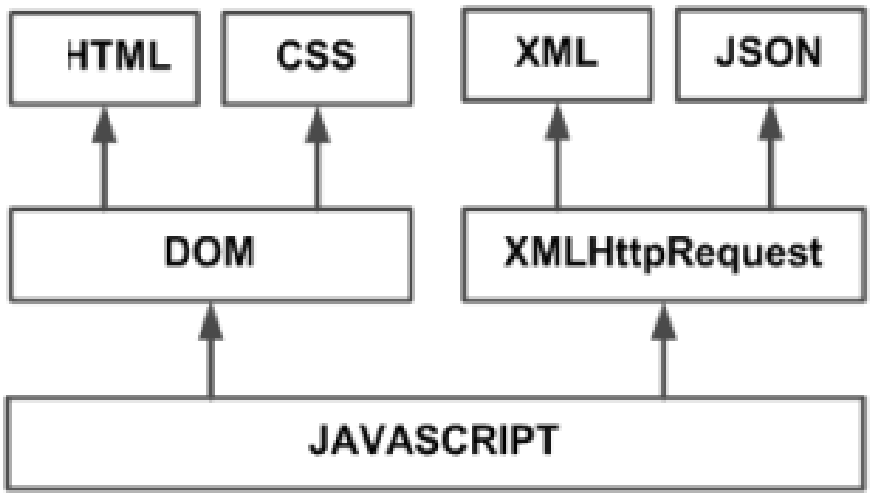

Figura 3.3: Tecnologías AJAX

3.2.1. XMLHttpRequest

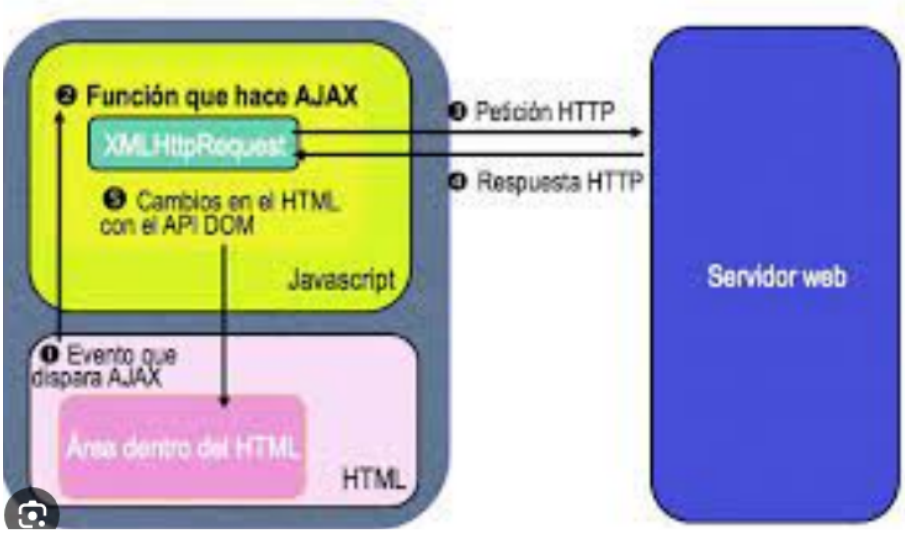

Figura 3.4: Modelo AJAX

XMLHttpRequest es un objeto del navegador que permite hacer solicitudes HTTP desde JavaScript.

Para trabajar con XMLHttpRequest y hacer una petición, necesitamos varios pasos:

1. **Crear** el objeto XMLHttpRequest

```
1
2 let xhr = new XMLHttpRequest();
```

2. **Inicializarlo**, normalmente justo después de new.

```
1
2 xhr.open(metodo, URL, [async, user, password])
```

Este método open establece los principales parámetros para la petición

método HTTP. (GET o POST).

URL a solicitar. Una URL (Uniform Resource Locator) es la dirección que identifica un recurso en la web, como una página, una imagen, un vídeo o un archivo.

*GET para **obtener** información del servidor y POST (crear o modificar) para **enviarla***

```
1
2 https://www.ejemplo.com:443/ruta/pagina.html?nombre=juan&id=5#seccion2
```

async si se asigna explícitamente a false, entonces la petición será sincrona. async = true (asíncrono) es el valor por defecto y el más usado.

user, password usuario y contraseña para autenticación (solo si es necesario).

La llamada al método open, contrario a su nombre, **no abre la conexión**. Solo **configura la solicitud**, pero la actividad de red solo empieza con la llamada del método send.

3. **Enviar.**

```
1
2 xhr.send([body])
```

Este método abre la conexión y envía la solicitud al servidor. El parámetro adicional body contiene el cuerpo de la solicitud.

Algunos métodos como GET no tienen cuerpo. Y otros como POST usan el parámetro body para enviar datos al servidor.

4. **Escuchar** los eventos de respuesta xhr. Los principales eventos son:
   - **onreadystatechange** - se dispara cada vez que cambia la propiedad readyState.

Valor `readyState`	Constante (no oficial en todos los navegadores)	Significado
0	`UNSENT`	El objeto `XMLHttpRequest` ha sido creado, pero aún no se llamó a `open()`.
1	`OPENED`	Se llamó a `open()`, la petición está configurada pero aún no se ha enviado.
2	`HEADERS_RECEIVED`	Se enviaron los encabezados de la petición y se recibieron los de respuesta.
3	`LOADING`	El cuerpo de la respuesta se está recibiendo (parcialmente descargado).
4	`DONE`	La operación terminó (con éxito o con error).

Figura 3.5: Valores de la propiedad readyState

Hay otra propiedad importante (xhr.status) que guardar el código generado por el protocolo http.

Rango de códigos	Nombre del rango	Significado general	Ejemplos comunes
1xx	Informativos	El servidor recibió la petición y el cliente debe continuar.	100 Continue, 101 Switching Protocols
2xx	Éxito	La petición fue recibida, entendida y procesada con éxito.	200 OK, 201 Created, 204 No Content
3xx	Redirecciones	Se requiere acción adicional para completar la petición.	301 Moved Permanently, 302 Found, 304 Not Modified
4xx	Errores del cliente	La petición tiene errores o no puede cumplirse por parte del cliente.	400 Bad Request, 401 Unauthorized, 403 Forbidden, 404 Not Found
5xx	Errores del servidor	El servidor falló al procesar una petición aparentemente válida.	500 Internal Server Error, 502 Bad Gateway, 503 Service Unavailable

Figura 3.6: Códigos http

- **onload** - cuando la petición se completa correctamente.
- **onerror** - si ocurre un error de red.
- **onabort** - si la petición fue cancelada con xhr.abort().
- **ontimeout** - si expira el tiempo máximo definido con xhr.timeout.
- **onprogress** - durante la descarga, se dispara varias veces con información del progreso.
- **onloadstart** - al empezar la petición.
- **onloadend** - al finalizar, sea con éxito o error.

Ejemplo:

```
1
2  <!DOCTYPE html>
3  <html lang="es">
4  <head>
5    <meta charset="UTF-8">
6    <title>GET y POST con XMLHttpRequest</title>
7  </head>
8  <body>
9    <script>
10     // GET: leer datos.json
11     function leerJSON() {
12       var xhr = new XMLHttpRequest();
13       xhr.open("GET", "https://jsonplaceholder.typicode.com/posts",
             true);
14
15       xhr.onreadystatechange = function () {
16         if (xhr.readyState === 4 && xhr.status === 200) {
17           alert("Respuesta "+ xhr.responseText);
18
19           // Parseamos el JSON
20           var datos = JSON.parse(xhr.responseText);
21         }
22       };
23
24       xhr.send();
25     }
26
27     // POST: enviar los datos a un endpoint
28     const data = JSON.stringify({
29       title: "Nuevo post de prueba",
30       body: "Este es el contenido del post",
31       userId: 1
32     });
33
34     function enviarPOST() {
35       // Crear la instancia del objeto XMLHttpRequest
36       const xhr = new XMLHttpRequest();
37
38       // Configurar la petición POST al endpoint de posts
39       xhr.open("POST", "https://jsonplaceholder.typicode.com/posts",
             true);
40
41       // Establecer el encabezado para indicar que el cuerpo es JSON
```

```
42        xhr.setRequestHeader("Content-Type", "application/json;
              charset=UTF-8");
43
44        // Definir qué hacer cuando llega la respuesta
45        xhr.onload = function() {
46        if (xhr.status === 201) { // 201 Created
47            const nuevoPost = JSON.parse(xhr.responseText);
48            alert("Post creado exitosamente:" + nuevoPost);
49        } else {
50            alert("Error al crear el post:" + xhr.status +
                  xhr.statusText);
51        }
52      };
53
54      // Enviar la petición con los datos
55      xhr.send(data);
56      }
57
58      // Ejecutamos el flujo: primero GET, luego POST
59      leerJSON();
60      enviarPOST();
61    </script>
62  </body>
63  </html>
```

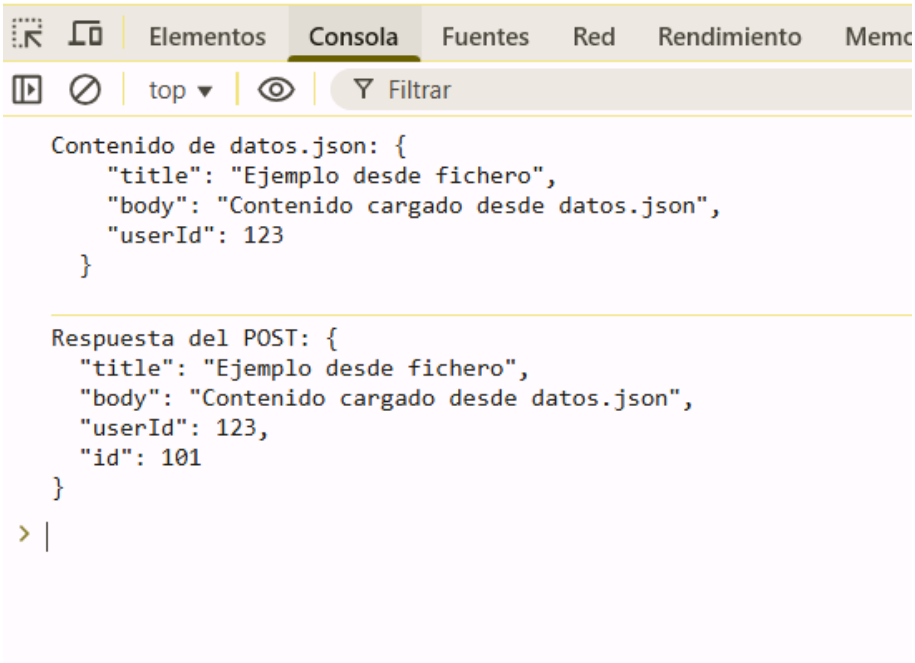

Figura 3.7: Ejecución del ejemplo

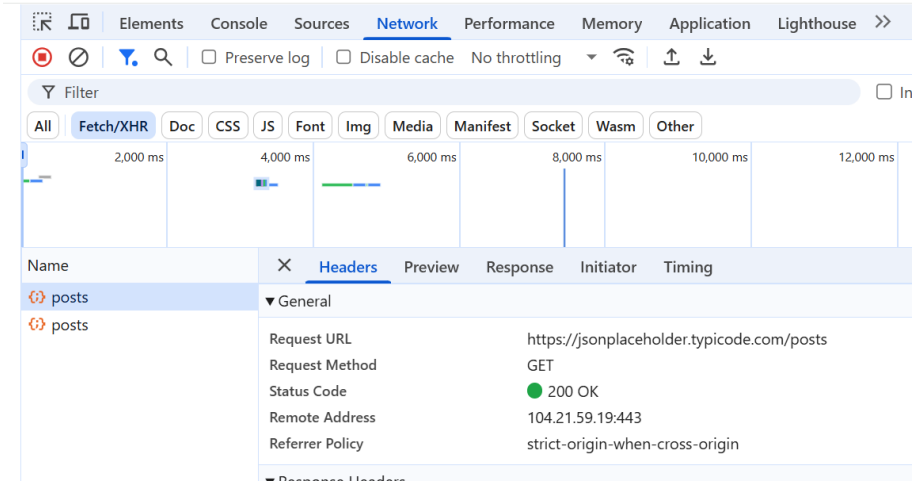

Figura 3.8: Herramientas de desarrolladores - Red (todo)

Figura 3.9: Herramientas de desarrolladores - Red (xhr/Fetch)

XMLHttpRequest ha sido la forma clásica de hacer peticiones AJAX desde los años 2000. Hoy en día, en casi todos los proyectos modernos se prefiere fetch().

Las razones principales son:

- Simplicidad y legibilidad.

```
1
2  var xhr = new XMLHttpRequest();
3  xhr.open("GET", "/api/data");
4  xhr.onload = function() {
5    if (xhr.status === 200) console.log(xhr.responseText);
6  };
7  xhr.send();
8
9
10
11 // Con fetch
12 fetch("/api/data")
13   .then(res => res.json())
14   .then(data => console.log(data));
```

- Manejo moderno de respuestas.
  Xhr devuelve todo como texto en responseText y tienes que convertir manualmente a JSON.
  fetch() tiene métodos como .json().
- Soporte para async/await.
  Con fetch puedes escribir código asincrónico que parece síncrono:

```
1
2  async function cargar() {
3    try {
4      let res = await fetch("/api/data");
5      let datos = await res.json();
6      console.log(datos);
7    } catch (e) {
8      console.error("Error:", e);
9    }
10 }
```

  Con XHR esto requiere mucho más código.
- Mejor soporte de APIs modernas.
- Etc...

XHR hoy todavía es importante por dos motivos: el primero es el soporte para navegadores antiguos, donde la **compatibilidad** de XHR es casi universal. El segundo, y más importante, es la necesidad de un seguimiento preciso del **progreso de transferencia** de datos. Mientras que Fetch requiere el uso de API más complejas como AbortController

o la manipulación de ReadableStreams para simular el progreso, XHR lo ofrece de forma nativa a través del evento progress y el objeto ProgressEvent.

3.3. API fetch

Como ya hemos comentado, la necesidad de manejar comunicaciones asíncronas de manera más legible y simple, llevó al desarrollo de patrones de diseño más limpios.

La arquitectura de comunicación en JavaScript se ha transformado con la adopción generalizada del modelo de **Promises** y la sintaxis **async/await.**

```
1
2  // Estructura básica de una promesa
3  const promesa = new Promise((resolve, reject) => {
4    // Aquí va el código asíncrono
5    // Si todo sale bien:
6    resolve(valor);
7    // Si algo falla:
8    reject(error);
9  });
```

Una **promesa** representa un valor que puede no estar disponible todavía. Al llamar a APIs modernas como Fetch, se devuelve inmediatamente una promesa que se resolverá (éxito) o se rechazará (error). Esto permite encadenar operaciones secuenciales con el método .then() y utilizar la sintaxis async/await para escribir código asíncrono que se asemeja al código síncrono, mejorando la legibilidad y la estructura de manejo de errores.

Cuando se llama a la función fetch(url, options), esta devuelve una promesa que se resuelve con un **objeto Response**, independientemente del estado HTTP.

Por defecto, la función fetch() realiza una solicitud GET, el método estándar para la lectura de recursos. Para realizar otro tipo de operaciones (no GET), es necesario configurar el objeto de opciones que se pasa como segundo argumento a fetch(). Este objeto debe incluir la propiedad method especificando el verbo HTTP deseado (post, put, delete).

```
1
2   <script>
3      fetch(url)
4        .then(response => {
5          if (!response.ok)
6          {
7            throw new Error("Error en la petición: " +
                 response.status);
8          }
9          return response.json(); // Convertimos la respuesta a JSON
10       })
11       .then(datos => {
```

```
12          console.log("Datos obtenidos:", datos);
13        })
14        .catch(error => {
15          console.error("Error:", error);
16        });
17    </script>
```

Explicación del ejemplo:

- fetch() devuelve una promesa
- Primer .then() maneja la respuesta HTTP y la convierte a JSON.
- Segundo .then() recibe los datos ya parseados.
- .catch() captura errores de red o si la respuesta no fue OK.

Después de que la promesa inicial de fetch() se resuelve con un objeto Response, el cuerpo de la respuesta aún no está disponible directamente. Tenemos que obtener y transformar el contenido del cuerpo utilizando métodos especializados.

El método **response.json** toma el cuerpo de la respuesta y lo analiza, deserializando la cadena JSON en un objeto o array nativo de JavaScript.

El método **response.text()** se utiliza cuando se espera que el cuerpo de la respuesta sea una cadena de texto simple. Esto es útil para recibir respuestas HTML o mensajes de error legibles.

La manipulación de recursos multimedia o archivos requiere el manejo de datos binarios. Fetch ofrece dos métodos para este propósito:

1. Lectura como Blob: **response.blob()** devuelve una promesa que se resuelve con un objeto Blob (Binary Large Object)
2. Lectura como ArrayBuffer: **response.arrayBuffer()** devuelve un ArrayBuffer. Se usan para audios, y procesamiento de bytes de bajo nivel.

```
1
2  // Ejemplo get
3
4  // Hacer una petición GET con fetch
5  fetch("https://jsonplaceholder.typicode.com/users")
6    .then(response => {
7      // Verificar si la respuesta fue exitosa
8      if (!response.ok) {
9        throw new Error("Error en la petición: " + response.status);
10     }
11     // Convertir la respuesta a JSON
12     return response.json();
13   })
14   .then(data => {
15     // Aquí 'data' es un array de usuarios
16     console.log("Lista de usuarios:", data);
17   })
18   .catch(error => {
19     // Capturar errores de red o de la petición
20     console.error("Hubo un problema con la petición:", error);
21   });
```

```
1
2  // Ejemplo post, para enviar datos al servidor
3
4  <script>
5      // Datos a enviar
6      const nuevoPost = {
7        title: "Mi primer post",
8        body: "Este es el contenido del post",
9        userId: 1
10     };
11
12     fetch("https://jsonplaceholder.typicode.com/posts", {
13       method: "POST", // Indicamos que es POST
14       headers: {
15         "Content-Type": "application/json" // Tipo de contenido
16       },
17       body: JSON.stringify(nuevoPost) // Convertimos el objeto a JSON
18     })
19     .then(response => {
20       if (!response.ok) {
21         throw new Error("Error en la petición: " + response.status);
22       }
23       return response.json();
24     })
25     .then(data => {
26       console.log("POST exitoso:", data);
27     })
28     .catch(error => {
29       console.error("Error:", error);
30     });
31   </script>
```

Explicación del ejemplo:

- method: POST - indica que queremos enviar datos al servidor.
- headers - especificamos que enviamos JSON.
- body - el contenido que enviamos, convertido a JSON.
  Para enviar datos estructurados de un objeto JavaScript al servidor, el objeto debe ser serializado en una cadena de texto JSON.
- response.json() - parsea la respuesta.
- .catch() - captura errores de red o HTTP.

La configuración de las cabeceras es imprescindible para comunicar al servidor el formato de los datos que se están enviando (el cuerpo de la petición) y las expectativas del cliente. Si esta cabecera se omite y se envía una cadena de texto, el navegador podría utilizar el valor por defecto *text/plain;charset=UTF-8*, lo cual impediría que el servidor interprete correctamente los datos JSON.

Para manejar la subida de archivos o la codificación de datos de formulario complejos, se utiliza la interfaz **FormData**. Un objeto FormData puede contener pares clave-valor que representan campos de formulario, y puede incluir objetos Blob (datos binarios) o File (que extienden Blob) para la transferencia de datos binarios, como imágenes o documentos.

Ejemplo de envío de un fichero:

```
1
2  <!DOCTYPE html>
3  <html lang="es">
4  <head>
5    <meta charset="UTF-8">
6    <title>Enviar archivo con fetch</title>
7  </head>
8  <body>
9
10   <h2>Subir archivo</h2>
11   <input type="file" id="archivoInput" />
12   <button id="enviarBtn">Enviar</button>
13
14   <script>
15     const inputArchivo = document.getElementById("archivoInput");
16     const botonEnviar = document.getElementById("enviarBtn");
17
18     botonEnviar.addEventListener("click", () => {
19       const archivo = document.getElementById("archivoInput").value;
20       if (!archivo)
21       {
22         alert("Selecciona un archivo primero");
23       } else {
24         // Creamos FormData y agregamos el archivo
25         const formData = new FormData();
26         formData.append("archivo", archivo);
27
28         // Enviamos al servidor
29         fetch("https://jsonplaceholder.typicode.com/posts", {
30           method: "POST",
31           body: formData // No necesitamos Content-Type, fetch lo
                 gestiona
32         })
33           .then(response => {
34             if (!response.ok) throw new Error("Error en la subida: "
                     + response.status);
35             return response.json();
36           })
37           .then(data => {
38             console.log("Archivo enviado con éxito:", data);
39           })
40           .catch(error => {
41             console.error("Error:", error);
42           });
43       }
44     });
45   </script>
46
47 </body>
```

```
48 </html>
```

Se puede mostrar el progreso de subida mostrando un porcentaje mientras se envía el fichero, usando XHR porque fetch aún no tiene soporte nativo para progreso de subida. Esto sirve para archivos grandes.

Un objeto FormData se puede construir directamente a partir de un elemento form existente en el DOM, o crearse manualmente para añadir campos específicos.

```
1
2 // Captura todos los campos del formulario
3 const formData = new FormData(form);
```

```
1
2  // Ejemplo con PUT. Se usa para modificar
3
4    <script>
5      const datosActualizados = {
6        title: "Título actualizado",
7        body: "Contenido actualizado del post",
8        userId: 1
9      };
10
11     fetch("https://jsonplaceholder.typicode.com/posts/1", { // ID
           del recurso
12       method: "PUT", // Método PUT
13       headers: {
14         "Content-Type": "application/json"
15       },
16       body: JSON.stringify(datosActualizados)
17     })
18     .then(response => {
19       if (!response.ok) {
20         throw new Error("Error en la actualización: " +
               response.status);
21       }
22       return response.json();
23     })
24     .then(data => {
25       console.log("Recurso actualizado:", data);
26     })
27     .catch(error => {
28       console.error("Error:", error);
29     });
30   </script>
```

Método	Acción principal	URL típica	ID asignado	Datos enviados
POST	Crear un nuevo recurso	`/posts`	Sí, el servidor asigna	Cuerpo con los datos
PUT	Actualizar/reemplazar un recurso	`/posts/1`	No, ya existe	Cuerpo con los datos completos

Figura 3.10: Diferencias entre post y put

```
1
2  // Ejemplo con DELETE. Se usa para eliminar
3
4  <!DOCTYPE html>
5  <html lang="es">
6  <head>
7    <meta charset="UTF-8">
8    <title>Ejemplo DELETE con fetch</title>
9  </head>
10 <body>
11   <script>
12     fetch("https://jsonplaceholder.typicode.com/posts/1", {
13       method: "DELETE" // Indicamos que queremos eliminar
14     })
15     .then(response => {
16       if (response.ok) {
17         console.log("Recurso eliminado correctamente");
18       } else {
19         console.error("Error al eliminar:", response.status);
20       }
21     })
22     .catch(error => {
23       console.error("Error de red:", error);
24     });
25   </script>
26 </body>
27 </html>
```

3.3.1. Async / await

La sintaxis async/await, introducida en ECMAScript 2017, es un avance en la forma de manejar la asincronía en JavaScript. Ofrece una estructura de código que simula el flujo síncrono.

Una función declarada con el prefijo **async** se convierte en una función asíncrona que, por definición, siempre retorna una promesa.

El operador **await** solo puede utilizarse dentro de una función async. Su propósito es pausar la ejecución del código dentro de esa función hasta que la promesa a la que se aplica se resuelva. Si la promesa se cumple, await devuelve el valor resuelto; si rechaza, lanza una excepción, permitiendo que sea capturada por un bloque try...catch.

La capacidad de await para pausar la función contenedora async es la principal razón de su legibilidad superior.

Otro de los beneficios de async/await es la posibilidad de utilizar la estructura síncrona try...catch de JavaScript para manejar errores y rechazos de promesas. Esto alinea la gestión de la asincronía con los patrones de código más familiares, mejorando la comprensión general del flujo de control.

```
1
2  <!DOCTYPE html>
3  <html lang="es">
4  <head>
5    <meta charset="UTF-8">
6    <title>Await/fetch</title>
7  </head>
8  <body>
9
10
11   <script>
12   async function loadJsonAsync(url)
13   {
14     try
15     {
16         const response = await fetch(url);
17
18
19         if (!response.ok)
20         {
21             // response.ok es true para códigos 200-299
22             throw new Error(`Response status: ${response.status}`);
23         }
24
25         const result = await response.json();
26
27         return result;
28
29     }
30     catch (error)
```

```
31      {
32
33          console.error("Error capturado:", error.message);
34          throw error;
35      }
36      }
37
38  // Uso
39  loadJsonAsync('https://javascript.info/no-such-user.json')
40   .catch(alert("Error")); // Error: 404
41
42    </script>
43  </body>
44  </html>
```

Los ejemplos vistos con .then, ahora con async/await

```
1
2  <script>
3    // Definimos una función asíncrona
4    async function obtenerDatos() {
5      const url = "https://jsonplaceholder.typicode.com/posts";
6
7      try {
8        const response = await fetch(url);
9
10       if (!response.ok) {
11         throw new Error("Error en la petición: " + response.status);
12       }
13
14       const datos = await response.json(); // Esperamos la
             conversión a JSON
15       console.log("Datos obtenidos:", datos);
16     } catch (error) {
17       console.error("Error:", error);
18     }
19   }
20
21   // Llamamos a la función
22   obtenerDatos();
23 </script>
```

```
1
2  <script>
3    // Definimos una función asíncrona
4    async function enviarPost() {
5      // Datos a enviar
6      const nuevoPost = {
7        title: "Mi primer post",
8        body: "Este es el contenido del post",
9        userId: 1
10     };
11
12     try {
13       // Enviamos la solicitud POST
14       const response = await
             fetch("https://jsonplaceholder.typicode.com/posts", {
15         method: "POST",
16         headers: {
17           "Content-Type": "application/json"
18         },
19         body: JSON.stringify(nuevoPost)
20       });
21
22       // Verificamos si la respuesta fue exitosa
23       if (!response.ok) {
24         throw new Error("Error en la petición: " + response.status);
25       }
26
27       // Esperamos y convertimos la respuesta a JSON
28       const data = await response.json();
29
30       console.log("POST exitoso:", data);
31     } catch (error) {
```

```
32        console.error("Error:", error);
33      }
34    }
35
36    // Llamamos a la función
37    enviarPost();
38  </script>
```

```
1
2   <!DOCTYPE html>
3   <html lang="es">
4   <head>
5     <meta charset="UTF-8">
6     <title>Enviar archivo con fetch y async/await</title>
7   </head>
8   <body>
9
10    <h2>Subir archivo</h2>
11    <input type="file" id="archivoInput" />
12    <button id="enviarBtn">Enviar</button>
13
14    <script>
15      const inputArchivo = document.getElementById("archivoInput");
16      const botonEnviar = document.getElementById("enviarBtn");
17
18      botonEnviar.addEventListener("click", async () => {
19        const archivo = inputArchivo.files[0]; // Tomamos el primer
              archivo
20
21        if (!archivo) {
22          alert("Selecciona un archivo primero");
23          return;
24        }
25
26        // Creamos el FormData y agregamos el archivo
27        const formData = new FormData();
28        formData.append("archivo", archivo);
29
30        try {
31          // Enviamos el archivo con fetch (POST)
32          const response = await
                fetch("https://jsonplaceholder.typicode.com/posts", {
33            method: "POST",
34            body: formData
35          });
36
37          if (!response.ok) {
38            throw new Error("Error en la subida: " + response.status);
39          }
40
41          const data = await response.json();
42          console.log("Archivo enviado con éxito:", data);
43          alert("Archivo enviado con éxito ?");
44        } catch (error) {
45          console.error("Error:", error);
46          alert("? Hubo un error al enviar el archivo");
47        }
48      });
49    </script>
```

```
50
51 </body>
52 </html>
```

```

1
2  <script>
3    // Definimos una función asíncrona
4    async function actualizarPost() {
5      const datosActualizados = {
6        title: "Título actualizado",
7        body: "Contenido actualizado del post",
8        userId: 1
9      };
10
11     try {
12       const response = await
             fetch("https://jsonplaceholder.typicode.com/posts/1", {
13         method: "PUT",
14         headers: {
15           "Content-Type": "application/json"
16         },
17         body: JSON.stringify(datosActualizados)
18       });
19
20       if (!response.ok) {
21         throw new Error("Error en la actualización: " +
               response.status);
22       }
23
24       const data = await response.json();
25       console.log("Recurso actualizado:", data);
26       alert("? Recurso actualizado correctamente");
27     } catch (error) {
28       console.error("Error:", error);
29       alert("? Hubo un error al actualizar el recurso");
30     }
31   }
32
33   // Llamamos a la función
34   actualizarPost();
35 </script>
```

```

1
2  <!DOCTYPE html>
3  <html lang="es">
4  <head>
5    <meta charset="UTF-8">
6    <title>Ejemplo DELETE con fetch y async/await</title>
7  </head>
8  <body>
9
10   <script>
11     // Función asíncrona para eliminar un recurso
12     async function eliminarPost() {
13       try {
14         const response = await
               fetch("https://jsonplaceholder.typicode.com/posts/1", {
15           method: "DELETE"
```

```
16          });
17
18          if (response.ok) {
19            console.log("Recurso eliminado correctamente ?");
20            alert("Recurso eliminado correctamente");
21          } else {
22            console.error("Error al eliminar:", response.status);
23            alert("Error al eliminar el recurso ?");
24          }
25        } catch (error) {
26          console.error("Error de red:", error);
27          alert("Error de red ?");
28        }
29      }
30
31      // Llamamos a la función
32      eliminarPost();
33    </script>
34
35  </body>
36  </html>
```

Independientemente de si se utiliza .then o async/await, hay una limitación crítica de la API fetch que difiere de otras librerías HTTP populares como Axios. La promesa devuelta por fetch() solo **rechaza si ocurre un error de red**. Si el servidor responde correctamente, aunque sea con un **código de error HTTP** (por ejemplo, 404 Not Found, 401 Unauthorized, o 500 Internal Server Error), la promesa se cumple (resuelve), devolviendo el objeto Response. Este comportamiento significa que ni el .catch en la cadena de promesas ni el catch del bloque try...catch se activarán automáticamente ante errores como un 404.

Para lograr un buen manejo de errores, es obligatorio que, inmediatamente después de recibir el objeto Response, se verifique la propiedad response.ok o response.status. Deberemos lanzar una excepción explícitamente si el estado no es satisfactorio.

```
1
2     if (!response.ok) {
3       // Si el estado es 4xx o 5xx, se lanza el error, activando el
            catch
4       throw new Error(`Response status: ${response.status}`);
5     }
```

3.4. Axios

Axios es una librería de JavaScript que permite hacer solicitudes HTTP desde el navegador de forma sencilla.

Es una alternativa a fetch con ventajas como:

- Sintaxis más limpia y fácil de usar.
- Soporte automático para JSON.
- Manejo de errores más intuitivo.
- Permite cancelar solicitudes y configurar interceptores.
- Compatible con promesas y async/await.

Para poder usarla en html:

```
1
2 <script
      src="https://cdn.jsdelivr.net/npm/axios/dist/axios.min.js"></script>
```

Ejemplos de uso

```
1  // GET (obtener datos)
2
3
4  // Con .then
5  axios.get('https://jsonplaceholder.typicode.com/posts/1')
6    .then(response => {
7      console.log("Datos recibidos:", response.data);
8    })
9    .catch(error => {
10     console.error("Error:", error);
11   });
12
13 // Con async/await
14 async function obtenerPost() {
15   try {
16     const response = await
           axios.get('https://jsonplaceholder.typicode.com/posts/1');
17     console.log("Datos recibidos:", response.data);
18   } catch (error) {
19     console.error("Error:", error);
20   }
21 }
22
23 obtenerPost();
24
25
26 //POST (enviar datos)
27
28 const nuevoPost = {
29   title: "Mi post con Axios",
30   body: "Contenido del post",
31   userId: 1
```

```
};

// Con promesas
axios.post('https://jsonplaceholder.typicode.com/posts', nuevoPost)
  .then(response => console.log("POST exitoso:", response.data))
  .catch(error => console.error("Error:", error));

// Con async/await
async function enviarPost() {
  try {
    const response = await
        axios.post('https://jsonplaceholder.typicode.com/posts',
        nuevoPost);
    console.log("POST exitoso:", response.data);
  } catch (error) {
    console.error("Error:", error);
  }
}

enviarPost();


// PUT / DELETE

await axios.put('https://jsonplaceholder.typicode.com/posts/1', {
    title: "Actualizado" });


// DELETE (elimina)
await axios.delete('https://jsonplaceholder.typicode.com/posts/1');
```

🔷 **Diferencias clave con `fetch`**

Característica	Axios	Fetch
JSON automático	✅ parsea automáticamente	❌ hay que usar `response.json()`
Errores HTTP	✅ atrapados en `catch` automáticamente	❌ `fetch` solo lanza error por red
Cancelación de solicitudes	✅ con tokens	❌ complicado
Soporte Node.js	✅	❌ requiere `node-fetch`

Figura 3.11: Axios vs Fetch

3.5. https://jsonplaceholder.typicode.com/posts

https://jsonplaceholder.typicode.com/posts es un servicio web de pruebaque se usa para simular un backend real cuando se está aprendiendo a trabajar con APIs y peticiones HTTP.

No es un servidor que guarde datos de verdad; simplemente responde con datos simulados para que puedas probar tus solicitudes GET, POST, PUT, PATCH (modificación parcial) y DELETE sin necesidad de montar un servidor propio.

Características principales:

- Dominio: jsonplaceholder.typicode.com
  Servicio gratuito de pruebas para desarrolladores.
- Recurso: /posts
  Simula un conjunto de posts como si fuera un blog o una base de datos.
- Operaciones disponibles:
  - GET /posts - Devuelve todos los posts.
  - GET /posts/1 - Devuelve el post con id = 1.
  - POST /posts - Simula la creación de un nuevo post y devuelve un JSON con un id generado.
  - PUT /posts/1 - Simula reemplazar el post con id = 1.
  - PATCH /posts/1 - Simula actualizar parcialmente el post con id = 1.
  - DELETE /posts/1 - Simula la eliminación del post con id = 1.
  - Datos simulados:

```
1   {
2     "userId": 1,
3     "id": 1,
4     "title": "sunt aut facere repellat provident occaecati
          excepturi optio reprehenderit",
5     "body": "quia et suscipit\nsuscipit recusandae ..."
6   }
```