**Ajax**

Contenido

[Conceptos clave de AJAX: 2](#_Toc188958419)

[¿Cómo funciona AJAX? 2](#_Toc188958420)

[Ejemplo básico usando XMLHttpRequest: 3](#_Toc188958421)

[Ejemplo usando la API fetch (más moderna): 3](#_Toc188958422)

[Ventajas de AJAX: 3](#_Toc188958423)

[Desventajas de AJAX: 3](#_Toc188958424)

[¿Cuándo usar JSON.stringify()? 5](#_Toc188958425)

[Sintaxis 6](#_Toc188958426)

[Ejemplo básico 6](#_Toc188958427)

[Ejemplo con un arreglo 7](#_Toc188958428)

[Uso del parámetro reviver 7](#_Toc188958429)

[Notas importantes 7](#_Toc188958430)

[Comparación con JSON.stringify() 8](#_Toc188958431)

[Resumen 8](#_Toc188958432)

[Creación del objeto URL 8](#_Toc188958433)

[Estructura básica de una URL 9](#_Toc188958434)

[Propiedades principales del objeto URL 9](#_Toc188958435)

[Gestión de parámetros de consulta 10](#_Toc188958436)

[Métodos útiles del objeto URL 10](#_Toc188958437)

[Ejemplo práctico completo 11](#_Toc188958438)

[Usos comunes del objeto URL 11](#_Toc188958439)

[Características clave del objeto FormData 12](#_Toc188958440)

[Creación del objeto FormData 12](#_Toc188958441)

[Métodos principales de FormData 12](#_Toc188958442)

[Enviar datos con FormData 13](#_Toc188958443)

[Manejo de archivos en FormData 14](#_Toc188958444)

[Ventajas de FormData 15](#_Toc188958445)

[Ejemplo práctico: Crear un formulario dinámico 15](#_Toc188958446)

[1. Acción del usuario o evento en la página 16](#_Toc188958447)

[2. Creación de un objeto de solicitud AJAX 16](#_Toc188958448)

[3. Configuración de la solicitud 17](#_Toc188958449)

[4. Envío de la solicitud al servidor 17](#_Toc188958450)

[5. Procesamiento de la solicitud en el servidor 17](#_Toc188958451)

[6. Respuesta del servidor 17](#_Toc188958452)

[7. Manejo de la respuesta en el cliente 18](#_Toc188958453)

[8. Actualización de la interfaz de usuario (UI) 18](#_Toc188958454)

[Resumen del flujo 18](#_Toc188958455)

[1. Implementación clásica con XMLHttpRequest 19](#_Toc188958456)

[2. Implementación moderna con Fetch API 20](#_Toc188958457)

[Comparativa: XMLHttpRequest vs Fetch 21](#_Toc188958458)

[Conclusión 21](#_Toc188958459)

[Ejemplo: GET con parámetros y encabezados personalizados 21](#_Toc188958460)

[Explicación paso a paso 22](#_Toc188958461)

[Salida esperada 23](#_Toc188958462)

[Desglose del JSON: 23](#_Toc188958463)

[Notas adicionales 24](#_Toc188958464)

[Ejemplo: Petición GET con cabeceras personalizadas usando XMLHttpRequest 24](#_Toc188958465)

[Explicación paso a paso 25](#_Toc188958466)

[Salida esperada 26](#_Toc188958467)

[Notas importantes 26](#_Toc188958468)

1. **Introducción**

**AJAX (Asynchronous JavaScript and XML)** es una técnica de desarrollo web que permite a las aplicaciones o páginas web comunicarse con el servidor de forma asíncrona, es decir, sin necesidad de recargar la página completa. Esto mejora significativamente la experiencia del usuario al hacer que las páginas sean más rápidas y responsivas.

### Conceptos clave de AJAX:

1. **Asíncrono**: Las solicitudes al servidor no bloquean la interacción del usuario con la página. El usuario puede seguir utilizando la página mientras los datos se procesan en segundo plano.
2. **JavaScript**: Es el lenguaje utilizado para manejar las solicitudes y procesar las respuestas.
3. **XML** (o JSON): Aunque originalmente AJAX usaba XML para intercambiar datos, hoy en día es más común usar formatos más simples como JSON.

### ¿Cómo funciona AJAX?

El flujo típico de AJAX incluye los siguientes pasos:

1. Un evento del usuario (como hacer clic en un botón) activa una solicitud.
2. AJAX envía una solicitud al servidor utilizando el objeto XMLHttpRequest o la API fetch.
3. El servidor procesa la solicitud y devuelve una respuesta (en formato XML, JSON, HTML o texto plano).
4. JavaScript procesa la respuesta y actualiza dinámicamente partes específicas de la página web, sin recargarla por completo.

### Ejemplo básico usando XMLHttpRequest:

// Crear un objeto XMLHttpRequest

const xhr = new XMLHttpRequest();

// Configurar la solicitud (método, URL y si es asíncrona)

xhr.open('GET', 'https://jsonplaceholder.typicode.com/posts', true);

// Manejar la respuesta

xhr.onload = function () {

if (xhr.status === 200) {

const data = JSON.parse(xhr.responseText); // Parsear la respuesta JSON

console.log(data);

} else {

console.error('Error en la solicitud:', xhr.status);

}

};

// Enviar la solicitud

xhr.send();

### Ejemplo usando la API fetch (más moderna):

fetch('https://jsonplaceholder.typicode.com/posts')

.then(response => {

if (!response.ok) throw new Error(`Error: ${response.status}`);

return response.json(); // Parsear la respuesta JSON

})

.then(data => console.log(data))

.catch(error => console.error('Error en la solicitud:', error));

### Ventajas de AJAX:

* **Mejora la experiencia del usuario**: No es necesario recargar toda la página, lo que la hace más rápida y fluida.
* **Ahorra ancho de banda**: Solo se envían y reciben los datos necesarios.
* **Interactivo y dinámico**: Las páginas web pueden actualizarse en tiempo real.

### Desventajas de AJAX:

* **Complejidad adicional**: Introduce lógica adicional para manejar solicitudes y respuestas.
* **Problemas de accesibilidad**: Algunos usuarios con navegadores antiguos o configuraciones específicas pueden tener dificultades.
* **Dependencia de JavaScript**: Si el usuario tiene deshabilitado JavaScript, AJAX no funcionará.
* **Manejo de errores**: Puede ser complicado manejar errores y fallos en las solicitudes asíncronas.

En resumen, **AJAX** es una técnica poderosa que ha revolucionado el desarrollo web, haciendo posibles aplicaciones web modernas y altamente interactivas como Google Maps, Gmail y muchas otras.

1. **¿Qué es JSON?**

**JSON** (JavaScript Object Notation) es un formato de texto ligero y fácil de leer/escribir que se utiliza para intercambiar datos entre un cliente (como un navegador) y un servidor. Es independiente del lenguaje de programación y se basa en una estructura de clave-valor similar a los objetos de JavaScript.

#### Características principales:

1. **Legible por humanos**: Su sintaxis es simple y fácil de entender.
2. **Independiente del lenguaje**: Aunque está basado en la sintaxis de objetos de JavaScript, puede ser utilizado en la mayoría de los lenguajes de programación como Python, Java, PHP, etc.
3. **Ligero**: Su estructura es mínima, lo que facilita el intercambio eficiente de datos.

#### Estructura de JSON:

JSON tiene dos estructuras principales:

1. **Objeto**: Un conjunto de pares clave-valor delimitados por {}.
2. **Array**: Una lista ordenada de valores delimitada por [].

##### Ejemplo de un JSON:

{

"nombre": "Juan",

"edad": 30,

"esEstudiante": false,

"cursos": ["Matemáticas", "Ciencias", "Historia"],

"direccion": {

"ciudad": "Madrid",

"pais": "España"

}

}

* 1. **¿Qué es JSON.stringify()?**

El método JSON.stringify() convierte un objeto de JavaScript en una cadena de texto en formato JSON. Es útil para enviar datos al servidor o almacenarlos en un lugar donde solo se acepte texto, como en localStorage o una base de datos.

#### Sintaxis:

JSON.stringify(valor, replacer, espacio);

* **valor**: El valor (objeto, arreglo, etc.) que deseas convertir a JSON.
* **replacer** (opcional): Una función o arreglo que puede filtrar o transformar los datos antes de convertirlos.
* **espacio** (opcional): Un número o cadena que se usa para formatear la salida con sangría (útil para hacer el JSON legible).

#### Ejemplo básico:

const persona = {

nombre: "Juan",

edad: 30,

esEstudiante: false

};

const jsonString = JSON.stringify(persona);

console.log(jsonString);

// Salida: '{"nombre":"Juan","edad":30,"esEstudiante":false}'

#### Ejemplo con espacio:

const jsonString = JSON.stringify(persona, null, 2);

console.log(jsonString);

/\* Salida:

{

"nombre": "Juan",

"edad": 30,

"esEstudiante": false

}

\*/

#### Ejemplo con replacer:

const persona = {

nombre: "Juan",

edad: 30,

esEstudiante: false

};

// Solo incluir "nombre" en el JSON

const jsonString = JSON.stringify(persona, ["nombre"]);

console.log(jsonString);

// Salida: '{"nombre":"Juan"}'

### ¿Cuándo usar JSON?stringify()?

1. **Enviar datos al servidor**:

const datos = { usuario: "Juan", edad: 30 };

fetch('/api/guardar', {

method: 'POST',

body: JSON.stringify(datos),

headers: { 'Content-Type': 'application/json' }

});

1. **Guardar datos en el navegador**:

const usuario = { nombre: "Ana", preferencias: { tema: "oscuro" } };

localStorage.setItem("usuario", JSON.stringify(usuario));

En resumen:

* **JSON** es un formato para estructurar datos de manera sencilla.
* El método JSON.stringify() convierte un objeto o arreglo de JavaScript en una cadena de texto JSON, lo que es útil para transmitir datos entre sistemas o almacenarlos.
  1. **¿Qué es JSON.parse()?**

El método JSON.parse() convierte una cadena de texto en formato JSON en un objeto o estructura de datos de JavaScript. Es la operación inversa a **JSON.stringify()**.

Se utiliza principalmente para interpretar datos que se reciben como texto desde una fuente externa, como un servidor, un archivo, o datos almacenados en el navegador (por ejemplo, en localStorage).

### Sintaxis

JSON.parse(texto, reviver);

* **texto**: Es la cadena de texto JSON que deseas convertir a un objeto de JavaScript. Esta cadena debe cumplir con el formato JSON válido.
* **reviver** (opcional): Una función que puede transformar los valores devueltos durante el proceso de conversión.

### Ejemplo básico

const jsonString = '{"nombre": "Juan", "edad": 30, "esEstudiante": false}';

const objeto = JSON.parse(jsonString);

console.log(objeto);

// Salida: { nombre: 'Juan', edad: 30, esEstudiante: false }

console.log(objeto.nombre); // "Juan"

console.log(objeto.edad); // 30

### Ejemplo con un arreglo

const jsonString = '[1, 2, 3, 4]';

const arreglo = JSON.parse(jsonString);

console.log(arreglo);

// Salida: [1, 2, 3, 4]

console.log(arreglo[2]); // 3

### Uso del parámetro reviver

El parámetro reviver es una función que permite personalizar el proceso de conversión. Se ejecuta para cada clave y valor del JSON, permitiéndote filtrar o transformar los valores.

#### Ejemplo:

const jsonString = '{"nombre": "Juan", "edad": 30}';

const objeto = JSON.parse(jsonString, (clave, valor) => {

if (clave === "edad") {

return valor + 1; // Incrementar la edad en 1

}

return valor; // Dejar los demás valores sin cambios

});

console.log(objeto);

// Salida: { nombre: 'Juan', edad: 31 }

### Notas importantes

1. **Formato JSON válido**:  
   La cadena que se pasa a **JSON.parse()** debe ser un JSON válido. Si contiene errores, como comillas simples en lugar de dobles o una estructura mal formada, el método lanzará un error SyntaxError.

#### Ejemplo de error:

const jsonString = "{'nombre': 'Juan'}"; // Comillas simples incorrectas

const objeto = JSON.parse(jsonString); // Error: SyntaxError

#### Corrección:

const jsonString = '{"nombre": "Juan"}'; // Comillas dobles válidas

const objeto = JSON.parse(jsonString);

1. **Uso en datos almacenados**:  
   Si usas localStorage o sessionStorage, puedes recuperar datos en formato JSON y convertirlos de vuelta a objetos.

const jsonString = '{"nombre": "Ana", "edad": 25}';

localStorage.setItem("usuario", jsonString);

const datos = JSON.parse(localStorage.getItem("usuario"));

console.log(datos.nombre); // "Ana"

1. **Cuidado con datos no confiables**:  
   Nunca utilices JSON.parse() directamente en cadenas de texto de origen desconocido o no confiable, ya que podría abrir la puerta a ataques de seguridad.

### Comparación con JSON.stringify()

| Método | Función |
| --- | --- |
| JSON.stringify() | Convierte un objeto de JavaScript a una cadena JSON. |
| JSON.parse() | Convierte una cadena JSON a un objeto de JavaScript. |

### Resumen

* JSON.parse() es un método que convierte una cadena en formato JSON en un objeto o estructura de datos de JavaScript.
* Puedes personalizar el proceso con el parámetro opcional reviver.
* Úsalo para interpretar datos de texto JSON provenientes de servidores, archivos o almacenamiento local.

1. **Gestión de URLs y formularios**
   1. **Objeto URL**

**El objeto URL en JavaScript es una interfaz que proporciona métodos y propiedades útiles para manipular y analizar URLs (Uniform Resource Locators). Este objeto permite descomponer una URL en sus partes, construir URLs dinámicamente y gestionar parámetros de consulta de manera sencilla.**

### ****Creación del objeto URL****

Puedes crear una instancia del objeto URL de las siguientes maneras:

#### ****1. A partir de una cadena de texto completa****

const url = new URL("https://www.ejemplo.com/carpeta/pagina.html?usuario=juan&id=123");

#### ****2. Usando un parámetro base (URL relativa)****

const url = new URL("/carpeta/pagina.html", "https://www.ejemplo.com");

### ****Estructura básica de una URL****

Una URL tiene varios componentes que el objeto URL permite analizar fácilmente:

https://usuario:contraseña@www.ejemplo.com:8080/carpeta/pagina.html?usuario=juan&id=123#seccion

\\_\_\_\_\_/ \\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_/ \\_\_\_\_\_/ \\_\_\_\_\_\_\_\_\_/ \\_\_\_\_\_/ \\_\_\_\_\_\_\_\_/ \\_\_\_\_\_/

esquema credenciales host puerto ruta query hash

### ****Propiedades principales del objeto**** URL

Una vez creada una instancia del objeto URL, puedes acceder a sus partes individuales mediante sus propiedades.

#### ****Propiedades comunes****

const url = new URL("https://usuario:contraseña@www.ejemplo.com:8080/carpeta/pagina.html?usuario=juan&id=123#seccion");

// Esquema o protocolo (ej. http, https)

console.log(url.protocol); // "https:"

// Nombre del host (incluye dominio y subdominio)

console.log(url.host); // "www.ejemplo.com:8080"

// Solo el dominio (sin el puerto)

console.log(url.hostname); // "www.ejemplo.com"

// Puerto (si está especificado)

console.log(url.port); // "8080"

// Ruta (después del dominio)

console.log(url.pathname); // "/carpeta/pagina.html"

// Cadena de consulta completa (query string)

console.log(url.search); // "?usuario=juan&id=123"

// Fragmento o hash (parte después del #)

console.log(url.hash); // "#seccion"

// Nombre de usuario y contraseña (si existen)

console.log(url.username); // "usuario"

console.log(url.password); // "contraseña"

// Origen completo (protocolo + host + puerto)

console.log(url.origin); // "https://www.ejemplo.com:8080"

### ****Gestión de parámetros de consulta****

El objeto URL incluye el método searchParams, que permite gestionar los parámetros de consulta (query string) de manera fácil y dinámica.

#### ****Ejemplo de manejo básico****

const url = new URL("https://www.ejemplo.com/pagina.html?usuario=juan&id=123");

// Obtener todos los parámetros

console.log(url.searchParams.toString()); // "usuario=juan&id=123"

// Obtener un valor específico

console.log(url.searchParams.get("usuario")); // "juan"

// Agregar un nuevo parámetro

url.searchParams.append("edad", "25");

console.log(url.toString()); // "https://www.ejemplo.com/pagina.html?usuario=juan&id=123&edad=25"

// Modificar un parámetro existente

url.searchParams.set("usuario", "carlos");

console.log(url.toString()); // "https://www.ejemplo.com/pagina.html?usuario=carlos&id=123&edad=25"

// Eliminar un parámetro

url.searchParams.delete("id");

console.log(url.toString()); // "https://www.ejemplo.com/pagina.html?usuario=carlos&edad=25"

// Comprobar si existe un parámetro

console.log(url.searchParams.has("edad")); // true

console.log(url.searchParams.has("id")); // false

### ****Métodos útiles del objeto**** URL

#### toString() ****y**** toJSON()

Ambos métodos devuelven la URL completa como una cadena de texto.

const url = new URL("https://www.ejemplo.com/pagina.html?usuario=juan");

console.log(url.toString()); // "https://www.ejemplo.com/pagina.html?usuario=juan"

console.log(url.toJSON()); // "https://www.ejemplo.com/pagina.html?usuario=juan"

#### URL.createObjectURL()

Crea una URL temporal que representa un archivo o un objeto en memoria, generalmente un Blob.

En JavaScript, un **Blob** (abreviatura de Binary Large Object) es un objeto que representa datos binarios en bruto de un archivo o de otro tipo de información que no necesariamente está en formato de texto. Es ideal para trabajar con datos que no son de texto, como imágenes, videos, archivos PDF, etc.

Un objeto **Blob** permite manejar grandes volúmenes de datos binarios de manera eficiente sin necesidad de cargarlos completamente en memoria.

### Características principales:

1. **Datos inmutables**: El contenido de un Blob no se puede cambiar después de que se crea.
2. **Manipulación de archivos**: Es comúnmente utilizado con la API File, para manipular y crear archivos.
3. **Fragmentación**: Puedes dividir un Blob en partes más pequeñas usando métodos como slice.

### Cómo crear un Blob

Puedes crear un Blob utilizando el constructor Blob():

const blob = new Blob(["Hola, este es un texto dentro de un blob"], { type: "text/plain" });

console.log(blob);

#### Argumentos:

1. **Un array de datos**: Puede contener cadenas de texto, objetos ArrayBuffer, TypedArray o incluso otros Blob.
2. **Opciones (opcional)**: Un objeto con propiedades como type (el tipo MIME, como text/plain, image/png, etc.).

### Usos comunes

#### 1. ****Descargar un archivo:****

const blob = new Blob(["Este es el contenido de mi archivo"], { type: "text/plain" });

const url = URL.createObjectURL(blob); // Crea una URL para descargar el blob

const link = document.createElement("a");

link.href = url;

link.download = "miArchivo.txt";

document.body.appendChild(link);

link.click();

URL.revokeObjectURL(url); // Libera la URL

#### 2. ****Subir un archivo con la API File:****

const input = document.createElement("input");

input.type = "file";

input.onchange = (event) => {

const archivo = event.target.files[0];

console.log(archivo);

};

document.body.appendChild(input);

#### 3. ****Crear una imagen a partir de datos binarios:****

const blob = new Blob(["<h1>Hola desde Blob</h1>"], { type: "text/html" });

const url = URL.createObjectURL(blob);

const iframe = document.createElement("iframe");

iframe.src = url;

document.body.appendChild(iframe);

const blob = new Blob(["Hola mundo"], { type: "text/plain" });

const objectURL = URL.createObjectURL(blob);

console.log(objectURL); // Ejemplo: "blob:https://www.ejemplo.com/0e0f1-3h4j-..."

URL.revokeObjectURL(objectURL); // Libera la memoria asociada a esta URL

#### URL.revokeObjectURL()

Libera la memoria asociada a una URL creada con URL.createObjectURL.

### ****Ejemplo práctico completo****

Supongamos que queremos construir una URL para una API y gestionar parámetros dinámicamente.

const apiBase = "https://api.misitio.com";

const endpoint = "/usuarios";

const url = new URL(endpoint, apiBase);

// Agregar parámetros dinámicos

url.searchParams.append("nombre", "Carlos");

url.searchParams.append("edad", "30");

url.searchParams.append("pais", "España");

console.log(url.toString());

// Resultado: "https://api.misitio.com/usuarios?nombre=Carlos&edad=30&pais=España"

// Actualizar un parámetro

url.searchParams.set("edad", "35");

// Quitar un parámetro

url.searchParams.delete("pais");

console.log(url.toString());

// Resultado: "https://api.misitio.com/usuarios?nombre=Carlos&edad=35"

### ****Usos comunes del objeto**** URL

1. **Manipular rutas y parámetros en aplicaciones web.**
   * Generar URLs dinámicas para APIs.
   * Leer y escribir parámetros en la barra de direcciones del navegador.
2. **Construcción de enlaces.**
   * Crear enlaces dinámicos para recursos o navegación.
3. **Gestión de blobs y archivos.**
   * Crear URLs para descargar o previsualizar archivos en memoria.
4. **Validación y análisis de URLs.**
   * Asegurarse de que las URLs cumplen un formato correcto antes de usarlas.
   1. **El objeto FormData**

El objeto FormData en JavaScript es una herramienta poderosa para trabajar con formularios. Permite construir conjuntos de pares clave/valor que representan los datos de un formulario HTML o cualquier conjunto de datos, y enviarlos fácilmente mediante solicitudes AJAX, como con fetch o XMLHttpRequest.

### ****Características clave del objeto**** FormData

1. **Recoge automáticamente los datos de un formulario HTML.**
2. **Maneja datos de forma estructurada, incluyendo archivos.**
3. **Envía datos usando el formato** multipart/form-data**, ideal para subir archivos.**

### ****Creación del objeto**** FormData

Puedes crear un objeto FormData de dos maneras:

#### ****1. A partir de un formulario HTML****

Si tienes un formulario en tu HTML, puedes crear un objeto FormData automáticamente con los datos del formulario:

<form id="miFormulario">

<input type="text" name="nombre" value="Juan">

<input type="email" name="email" value="juan@example.com">

<input type="file" name="archivo">

</form>

const formulario = document.getElementById("miFormulario");

const formData = new FormData(formulario);

#### ****2. Crearlo vacío y añadir datos manualmente****

Si no tienes un formulario o deseas agregar datos manualmente, puedes hacerlo:

const formData = new FormData();

formData.append("nombre", "Juan");

formData.append("email", "juan@example.com");

formData.append("edad", 30);

### ****Métodos principales de**** FormData

#### ****1.**** append()

Añade un par clave/valor al objeto FormData.

formData.append("nombre", "Juan");

formData.append("archivo", fileInput.files[0]); // Agregar un archivo

#### ****2.**** set()

Similar a append(), pero si la clave ya existe, sobrescribe el valor.

formData.set("nombre", "Carlos"); // Sobrescribe el valor anterior

#### ****3.**** get()

Obtiene el valor asociado a una clave.

console.log(formData.get("nombre")); // "Carlos"

#### ****4.**** getAll()

Obtiene todos los valores asociados a una clave (útil si hay valores múltiples).

formData.append("intereses", "programación");

formData.append("intereses", "música");

console.log(formData.getAll("intereses")); // ["programación", "música"]

#### ****5. has()****

Verifica si existe una clave en el objeto FormData.

console.log(formData.has("nombre")); // true

console.log(formData.has("apellido")); // false

#### ****6.**** delete()

Elimina un par clave/valor del objeto FormData.

formData.delete("edad");

console.log(formData.has("edad")); // false

#### ****7. Iterar sobre los datos (****entries()****)****

Puedes usar el método entries() o un bucle for...of para iterar sobre los pares clave/valor del objeto FormData.

for (const [clave, valor] of formData.entries()) {

console.log(`${clave}: ${valor}`);

}

### ****Enviar datos con**** FormData

El objeto FormData se utiliza comúnmente para enviar datos al servidor mediante AJAX.

#### ****Ejemplo básico con**** fetch

const formData = new FormData();

formData.append("nombre", "Juan");

formData.append("archivo", fileInput.files[0]);

fetch("https://miapi.com/subir", {

method: "POST",

body: formData, // Enviar el FormData

})

.then(response => response.json())

.then(data => {

console.log("Respuesta del servidor:", data);

})

.catch(error => {

console.error("Error:", error);

});

#### ****Con un formulario HTML****

const formulario = document.getElementById("miFormulario");

formulario.addEventListener("submit", (event) => {

event.preventDefault(); // Evitar recargar la página

const formData = new FormData(formulario);

fetch("https://miapi.com/guardar", {

method: "POST",

body: formData,

})

.then(response => response.json())

.then(data => console.log("Datos enviados:", data))

.catch(error => console.error("Error:", error));

});

### ****Manejo de archivos en**** FormData

El objeto FormData es ideal para subir archivos, ya que automáticamente incluye los datos del archivo en la solicitud.

<form id="subirArchivo">

<input type="file" name="archivo">

<button type="submit">Subir</button>

</form>

const formulario = document.getElementById("subirArchivo");

formulario.addEventListener("submit", (event) => {

event.preventDefault();

const formData = new FormData(formulario);

fetch("https://miapi.com/subirArchivo", {

method: "POST",

body: formData,

})

.then(response => response.json())

.then(data => console.log("Archivo subido correctamente:", data))

.catch(error => console.error("Error al subir el archivo:", error));

});

### ****Ventajas de**** FormData

1. **Compatibilidad con archivos:** Maneja archivos fácilmente, incluidos varios a la vez.
2. **Codificación automática:** Los datos se codifican en formato multipart/form-data, adecuado para enviar formularios.
3. **Soporte dinámico:** Puedes agregar, eliminar o modificar datos en tiempo de ejecución.
4. **Compatibilidad amplia:** Funciona con la mayoría de las APIs modernas, como fetch y XMLHttpRequest.

### ****Ejemplo práctico: Crear un formulario dinámico****

Crea un formulario que permita subir un archivo junto con otros datos del usuario.

#### HTML:

<form id="formularioDinamico">

<label for="nombre">Nombre:</label>

<input type="text" id="nombre" name="nombre" required>

<label for="archivo">Archivo:</label>

<input type="file" id="archivo" name="archivo" required>

<button type="submit">Enviar</button>

</form>

<div id="resultado"></div>

#### JavaScript:

document.getElementById("formularioDinamico").addEventListener("submit", (event) => {

event.preventDefault();

const form = event.target;

const formData = new FormData(form);

fetch("https://miapi.com/guardarDatos", {

method: "POST",

body: formData,

})

.then(response => response.json())

.then(data => {

document.getElementById("resultado").innerText = "Datos enviados correctamente.";

console.log("Respuesta del servidor:", data);

})

.catch(error => {

document.getElementById("resultado").innerText = "Error al enviar los datos.";

console.error("Error:", error);

});

});

1. **Ajax de forma nativa**

El escenario habitual de una **petición AJAX** implica una comunicación asíncrona entre un cliente (normalmente un navegador web) y un servidor, utilizando JavaScript para enviar y recibir datos sin recargar la página. Aquí te lo explico paso a paso:

### ****1. Acción del usuario o evento en la página****

El proceso comienza con una acción que desencadena la petición AJAX. Por ejemplo:

* Un clic en un botón.
* La selección de una opción en un formulario.
* Un evento automático (como cargar la página o un temporizador).

### ****2. Creación de un objeto de solicitud AJAX****

El navegador crea un objeto para manejar la solicitud. Hay dos formas comunes:

* **XMLHttpRequest**: El método tradicional, aunque más **verboso**.
* **Fetch API**: Una opción más moderna y flexible.

En el contexto de programación, **"verboso"** se refiere a algo que requiere escribir más código o detalles de lo necesario para realizar una tarea, lo que puede hacer que el código sea más largo o complicado de leer.

**Ejemplo con XMLHttpRequest:**

let xhr = new XMLHttpRequest();

Ejemplo con Fetch:

fetch(url).then(response => response.json());

### ****3. Configuración de la solicitud****

El desarrollador configura detalles como:

* **El método HTTP**: GET, POST, PUT, DELETE, etc.
* **La URL del recurso**: El endpoint donde se enviará la solicitud.
* **Datos adicionales**: Parámetros, encabezados HTTP o cuerpo de la solicitud.

Con XMLHttpRequest:

xhr.open('GET', 'https://api.example.com/data', true);

Con Fetch:

fetch('https://api.example.com/data', {

method: 'GET'

});

### ****4. Envío de la solicitud al servidor****

Se envía la solicitud al servidor para procesarla.

Con XMLHttpRequest:

xhr.send();

Con Fetch:

fetch(url, options);

### ****5. Procesamiento de la solicitud en el servidor****

En el servidor:

* Se recibe la solicitud enviada por el cliente.
* El servidor procesa la solicitud (consulta a una base de datos, cálculos, etc.).
* El servidor genera una respuesta en formato compatible, como **JSON**, **XML** o **HTML**.

### ****6. Respuesta del servidor****

El servidor envía una respuesta al cliente, que contiene:

* **Datos solicitados** (por ejemplo, registros de una base de datos).
* **Mensajes de error** (en caso de problemas).
* **Estado HTTP** (por ejemplo, 200 OK para éxito o 404 Not Found para errores).

### ****7. Manejo de la respuesta en el cliente****

El cliente recibe la respuesta del servidor y la procesa:

* Si se usa XMLHttpRequest:

xhr.onload = function () {

if (xhr.status === 200) {

console.log(xhr.responseText); // Procesar la respuesta

}

};

* Si se usa Fetch:

fetch('https://api.example.com/data')

.then(response => response.json())

.then(data => console.log(data)) // Procesar la respuesta

.catch(error => console.error(error)); // Manejar errores

### ****8. Actualización de la interfaz de usuario (UI)****

Con los datos recibidos, el cliente actualiza dinámicamente la página web, como:

* Mostrar datos nuevos en una tabla.
* Mostrar un mensaje de éxito o error.
* Cambiar partes de la página sin recargarla.

### ****Resumen del flujo****

1. **Evento del usuario.**
2. **Creación del objeto AJAX (XMLHttpRequest o Fetch).**
3. **Configuración de la solicitud.**
4. **Envío al servidor.**
5. **Procesamiento en el servidor.**
6. **Respuesta del servidor.**
7. **Manejo de la respuesta.**
8. **Actualización de la UI.**
   1. **Implementación de una petición AJAX**

La implementación de una petición AJAX en JavaScript puede realizarse utilizando dos enfoques principales: el clásico con **XMLHttpRequest** o el moderno con la **Fetch API**. Aquí te explico ambos con ejemplos prácticos:

### ****1. Implementación clásica con XMLHttpRequest****

#### Paso a paso:

1. Crea un objeto XMLHttpRequest.
2. Configura el método HTTP y la URL del recurso con open().
3. Define la función de callback con onload o onreadystatechange para manejar la respuesta.
4. Envía la solicitud con send().

#### Ejemplo: Hacer una petición GET

// Crear el objeto XMLHttpRequest

let xhr = new XMLHttpRequest();

// Configurar la solicitud (método, URL y asíncrono)

xhr.open('GET', 'https://api.example.com/data', true);

// Manejar la respuesta cuando esté lista

xhr.onload = function () {

if (xhr.status === 200) {

console.log('Datos recibidos:', JSON.parse(xhr.responseText)); // Procesar la respuesta

} else {

console.error('Error en la solicitud:', xhr.status);

}

};

// Manejar errores de red

xhr.onerror = function () {

console.error('Error de red');

};

// Enviar la solicitud

xhr.send();

#### Ejemplo: Hacer una petición POST con datos

let xhr = new XMLHttpRequest();

xhr.open('POST', 'https://api.example.com/data', true);

// Configurar el encabezado para enviar JSON

xhr.setRequestHeader('Content-Type', 'application/json');

// Manejar la respuesta

xhr.onload = function () {

if (xhr.status === 201) {

console.log('Datos enviados con éxito:', JSON.parse(xhr.responseText));

} else {

console.error('Error en la solicitud:', xhr.status);

}

};

// Enviar los datos como JSON

xhr.send(JSON.stringify({ nombre: 'Juan', edad: 30 }));

### ****2. Implementación moderna con Fetch API****

La API fetch es más fácil de usar y trabaja con Promesas, lo que hace que el código sea más limpio y manejable.

#### Paso a paso:

1. Llama a fetch() con la URL y un objeto de configuración (opcional).
2. Procesa la respuesta usando .then() y .catch() para manejar errores.
3. Convierte los datos si es necesario (por ejemplo, a JSON).

#### Ejemplo: Hacer una petición GET

fetch('https://api.example.com/data')

.then(response => {

if (!response.ok) {

throw new Error(`Error en la solicitud: ${response.status}`);

}

return response.json(); // Convertir a JSON

})

.then(data => {

console.log('Datos recibidos:', data); // Procesar la respuesta

})

.catch(error => {

console.error('Hubo un error:', error);

});

#### Ejemplo: Hacer una petición POST con datos

fetch('https://api.example.com/data', {

method: 'POST', // Método HTTP

headers: {

'Content-Type': 'application/json' // Encabezado para enviar JSON

},

body: JSON.stringify({ nombre: 'Ana', edad: 25 }) // Datos a enviar

})

.then(response => {

if (!response.ok) {

throw new Error(`Error en la solicitud: ${response.status}`);

}

return response.json();

})

.then(data => {

console.log('Datos enviados con éxito:', data);

})

.catch(error => {

console.error('Hubo un error:', error);

});

### ****Comparativa: XMLHttpRequest vs Fetch****

|  |  |  |
| --- | --- | --- |
| **Característica** | **XMLHttpRequest** | **Fetch API** |
| Facilidad de uso | Verboso y más complicado | Más limpio y sencillo |
| Uso de Promesas | No (callback-based) | Sí, basado en Promesas |
| Manejador de errores | onerror y manejo manual | .catch() automático |
| Soporte para modernidad | Compatible con navegadores más antiguos | Requiere navegadores modernos |

### ****Conclusión****

Hoy en día, se recomienda usar la **Fetch API** debido a su simplicidad, compatibilidad con Promesas y facilidad de manejo de errores. Sin embargo, es importante conocer ambas opciones porque XMLHttpRequest aún se usa en proyectos heredados.

**Ejemplo de una petición GET al endpoint https://httpbin.org/get, enviando parámetros en la URL** y **encabezados personalizados**

Aquí tienes un ejemplo de una petición GET al endpoint https://httpbin.org/get, enviando **parámetros en la URL** y **encabezados personalizados**. También lo explico paso a paso:

### ****Ejemplo: GET con parámetros y encabezados personalizados****

// URL del endpoint con parámetros en la query string

const url = 'https://httpbin.org/get?param1=value1&param2=value2';

// Configuración de los encabezados personalizados

const headers = {

'Authorization': 'Bearer mi\_token\_seguro',

'Custom-Header': 'MiValorPersonalizado',

};

// Realizamos la petición GET con encabezados

fetch(url, {

method: 'GET', // Método HTTP

headers: headers, // Añadimos los encabezados personalizados

})

.then(response => {

// Verificamos si la respuesta fue exitosa

if (!response.ok) {

throw new Error(`Error en la solicitud: ${response.status}`);

}

return response.json(); // Convertimos la respuesta a JSON

})

.then(data => {

// Procesamos los datos recibidos

console.log('Datos recibidos de la API:', data);

})

.catch(error => {

// Manejo de errores

console.error('Hubo un error en la petición:', error);

});

### ****Explicación paso a paso****

1. **Definimos la URL con parámetros**

const url = 'https://httpbin.org/get?param1=value1&param2=value2';

* + Se añaden parámetros en la URL con una **query string**:
    - param1=value1
    - param2=value2
  + La query string comienza con ? y cada parámetro se separa con &.

1. **Configuramos los encabezados personalizados**

const headers = {

'Authorization': 'Bearer mi\_token\_seguro',

'Custom-Header': 'MiValorPersonalizado',

};

* + En este ejemplo, usamos:
    - Authorization: Un token ficticio para autorización.
    - Custom-Header: Un valor personalizado, que puedes usar para pruebas o comunicación especial con el servidor.

1. **Realizamos la petición GET**

fetch(url, {

method: 'GET', // Especificamos el método (opcional, ya que GET es por defecto)

headers: headers, // Pasamos los encabezados definidos

})

* + fetch() recibe un segundo argumento que es un objeto de configuración.
  + En este objeto configuramos:
    - method: El método HTTP, que en este caso es **GET**.
    - headers: Los encabezados personalizados que queremos incluir.

1. **Manejamos la respuesta y los errores**

.then(response => {

if (!response.ok) {

throw new Error(`Error en la solicitud: ${response.status}`);

}

return response.json(); // Convertimos la respuesta a JSON

})

.catch(error => {

console.error('Hubo un error en la petición:', error);

});

* + Verificamos si la respuesta es exitosa con response.ok.
  + Si hay un problema, lanzamos un error con el código de estado (response.status).
  + Convertimos la respuesta a un objeto JSON para procesarla más fácilmente.

### ****Salida esperada****

La API https://httpbin.org/get devolverá un objeto JSON que contiene:

{

"args": {

"param1": "value1",

"param2": "value2"

},

"headers": {

"Authorization": "Bearer mi\_token\_seguro",

"Custom-Header": "MiValorPersonalizado",

"Host": "httpbin.org",

"User-Agent": "Mozilla/5.0 (etc...)"

},

"origin": "203.0.113.45",

"url": "https://httpbin.org/get?param1=value1&param2=value2"

}

### Desglose del JSON:

* args: Muestra los parámetros que enviamos (param1 y param2).
* headers: Incluye los encabezados personalizados enviados (Authorization y Custom-Header).
* origin: Dirección IP de origen.
* url: La URL completa que procesó la solicitud.

### ****Notas adicionales****

* Puedes probar este código directamente en el navegador o en entornos como Node.js (con un paquete como node-fetch).
* La cabecera Authorization se utiliza comúnmente para autenticación, pero puede ser reemplazada con lo que necesites.

Para personalizar las cabeceras HTTP en las peticiones realizadas con XMLHttpRequest, puedes utilizar el método .setRequestHeader(). Este método te permite establecer cabeceras personalizadas antes de enviar la solicitud.

**Cómo personalizar las cabeceras HTTP en una solicitud con XMLHttpRequest.**

### ****Ejemplo: Petición GET con cabeceras personalizadas usando XMLHttpRequest****

// Crear un nuevo objeto XMLHttpRequest

const xhr = new XMLHttpRequest();

// Configurar la URL del endpoint

const url = 'https://httpbin.org/get';

// Abrir la solicitud GET

xhr.open('GET', url, true);

// Configurar cabeceras personalizadas

xhr.setRequestHeader('Authorization', 'Bearer mi\_token\_seguro');

xhr.setRequestHeader('Custom-Header', 'MiValorPersonalizado');

// Manejar el evento de cambio de estado

xhr.onreadystatechange = function () {

if (xhr.readyState === 4) { // 4 significa que la solicitud está completa

if (xhr.status >= 200 && xhr.status < 300) { // Verifica el código de estado HTTP

console.log('Respuesta del servidor:', JSON.parse(xhr.responseText));

} else {

console.error(`Error: ${xhr.status} - ${xhr.statusText}`);

}

}

};

// Manejar errores de red

xhr.onerror = function () {

console.error('Hubo un error en la conexión');

};

// Enviar la solicitud

xhr.send();

### ****Explicación paso a paso****

1. **Crear el objeto XMLHttpRequest**

const xhr = new XMLHttpRequest();

* + Este objeto nos permite realizar solicitudes HTTP desde JavaScript.

1. **Abrir la conexión con .open()**

xhr.open('GET', url, true);

* + Método: GET.
  + URL: https://httpbin.org/get.
  + true indica que la solicitud es **asíncrona**.

1. **Establecer cabeceras personalizadas con .setRequestHeader()**

xhr.setRequestHeader('Authorization', 'Bearer mi\_token\_seguro');

xhr.setRequestHeader('Custom-Header', 'MiValorPersonalizado');

* + **Authorization**: Una cabecera comúnmente utilizada para enviar tokens de autenticación.
  + **Custom-Header**: Ejemplo de una cabecera personalizada que puedes definir según las necesidades del servidor.
  + Este método debe llamarse después de xhr.open() pero antes de xhr.send().

1. **Manejar la respuesta**

xhr.onreadystatechange = function () {

if (xhr.readyState === 4) { // Comprobamos que la solicitud esté completa

if (xhr.status >= 200 && xhr.status < 300) {

console.log('Respuesta del servidor:', JSON.parse(xhr.responseText));

} else {

console.error(`Error: ${xhr.status} - ${xhr.statusText}`);

}

}

};

* + xhr.readyState === 4: La solicitud ha sido completada.
  + xhr.status: Código de estado HTTP devuelto por el servidor.
  + xhr.responseText: La respuesta del servidor en formato de texto plano. Si es JSON, puedes usar JSON.parse() para convertirlo a un objeto.

1. **Manejo de errores**

xhr.onerror = function () {

console.error('Hubo un error en la conexión');

};

* + xhr.onerror se ejecuta si ocurre un error de red, como problemas de conexión.

1. **Enviar la solicitud**

xhr.send();

* + Inicia la comunicación con el servidor.

### ****Salida esperada****

Si ejecutas este código, recibirás un objeto JSON similar al siguiente (simulado con https://httpbin.org/get):

{

"args": {}, // Parámetros enviados (vacío en este caso)

"headers": { // Encabezados enviados

"Authorization": "Bearer mi\_token\_seguro",

"Custom-Header": "MiValorPersonalizado",

"Host": "httpbin.org",

"User-Agent": "Mozilla/5.0 (etc...)"

},

"origin": "203.0.113.45", // Dirección IP del cliente

"url": "https://httpbin.org/get"

}

### ****Notas importantes****

1. **Orden correcto:**
   * Las cabeceras deben configurarse **después** de xhr.open() y **antes** de xhr.send().
2. **Sobreescritura de cabeceras estándar:**
   * Algunas cabeceras, como Content-Type, pueden ser sobreescritas, pero otras como Host no pueden ser modificadas debido a restricciones de seguridad.
3. **Asincronía:**
   * Al configurar xhr.open() con true, la solicitud se realiza de manera asíncrona. Esto significa que el resto del código no se bloquea mientras la solicitud se completa.