

Unidad 7 - AJAX

Introducción

AJAX (Asynchronous JavaScript And XML) es una técnica que permite interactuar con el servidor sin necesidad de recargar la página web. Con AJAX podemos:

- Realizar peticiones al servidor (con o sin envío de datos).
- Recibir y procesar datos de forma asíncrona.

Esto significa que la página no se bloquea mientras espera una respuesta, sino que los datos se manejan en segundo plano y se actualizan cuando llegan.

Aunque originalmente AJAX usaba XML para el intercambio de datos, hoy en día es más común emplear formatos como JSON o texto plano.

Evolución de las técnicas AJAX

El manejo de peticiones asíncronas ha evolucionado con el tiempo:

1. `XMLHttpRequest` → Introducido en Internet Explorer 5 (1998), fue la primera API para realizar peticiones AJAX.
2. **jQuery** (2006) → Facilitó el uso de AJAX con métodos simplificados como `$.ajax()`.
3. **Fetch API** (2015) → Se convirtió en el estándar moderno para realizar peticiones asíncronas de manera más eficiente y con mejor soporte en los navegadores.

Consideraciones clave

- **Entorno de desarrollo:** AJAX requiere un modelo cliente-servidor. Se puede trabajar en local con herramientas como XAMPP o WAMP, o subir los proyectos a un servidor web.
- **Recarga de página:** Para evitar problemas de caché, utiliza `Ctrl + F5` para forzar la actualización.
- **Política de seguridad:** Los navegadores aplican la política de mismo origen (*same-origin policy*), lo que significa que las peticiones AJAX solo pueden acceder a recursos del mismo dominio (protocolo + dominio + puerto).
- **Estudio de `XMLHttpRequest`:** Aunque ha sido reemplazada por `Fetch`, es importante conocer esta API para mantener proyectos que aún la utilicen.

El objeto `XMLHttpRequest`

Introducción

`XMLHttpRequest` es un objeto de JavaScript creado por Microsoft y adoptado por Mozilla, Apple y Google, convirtiéndose en un estándar del W3C.

Originalmente, esta API estaba diseñada para obtener datos en XML desde un servidor mediante HTTP. Sin embargo, versiones posteriores permitieron el uso de otros formatos como texto plano y JSON. Además de HTTP, puede funcionar con otros protocolos como `file://` o `ftp://`.

Cómo trabajar con `XMLHttpRequest`

Para realizar una petición AJAX con `XMLHttpRequest`, se siguen estos pasos:

1. Crear el objeto `XMLHttpRequest`

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

2. Inicializar la petición

```
xhr.open(method, url, [async, user, password]);
```

Parámetros:

- `method` → Método HTTP (`GET` o `POST` son los más comunes).
- `url` → URL del recurso solicitado.
- `async` (*opcional*) → `true` (por defecto) para peticiones asíncronas.
- `user` (*opcional*) → Nombre de usuario.
- `password` (*opcional*) → Contraseña.

```
xhr.open('GET', './datos.php?localidad=45');
```

3. Configurar las cabeceras HTTP (*opcional*)

```
xhr.setRequestHeader(cabecera, valor);
```

```
// Enviar, con "POST", los parámetros en pares clave=valor separados por "  
xhr.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
```

```
// Enviar, con "POST", un JSON codificado con UTF-8  
xhr.setRequestHeader("Content-Type", "application/json; charset=utf-8");
```

4. Definir el tipo de respuesta (*opcional*)

```
xhr.responseType = "json"; // Puede ser "", "text" o "json"
```

5. Manejar la respuesta del servidor

Para gestionar la respuesta del servidor, se define un `listener` que captura la respuesta cuando la solicitud se completa:

```
xhr.addEventListener("load", function () {  
  if (xhr.status !== 200) {  
    alert(`Error ${xhr.status}: ${xhr.statusText}`);  
  } else {  
    let data = xhr.response;  
    console.log(data);  
  }  
});
```

Propiedades clave del objeto `XMLHttpRequest` (sólo lectura):

<code>xhr.status</code>	Código de estado HTTP de la respuesta (ej: <code>200 OK</code> , <code>403 Forbidden</code> , <code>404 Not Found</code>).
<code>xhr.statusText</code>	Texto descriptivo del estado HTTP devuelto por el servidor.
<code>xhr.response</code>	Contenido de la respuesta en el formato especificado por <code>.responseType</code> (<code>ArrayBuffer</code> , <code>Blob</code> , <code>Document</code> , <code>Object</code> o <code>string</code>).
<code>xhr.responseText</code>	Contiene la respuesta en formato <code>string</code> (solo si la solicitud tuvo éxito y <code>responseType</code> no se cambió).
<code>xhr.responseXML</code>	Devuelve un objeto <code>Document</code> con la respuesta en XML o <code>null</code> si la conversión no fue posible.

6. Definir eventos adicionales (opcional)

Es recomendable definirlos antes de llamar a `xhr.send()`.

Eventos comunes:

- `error` → Se dispara cuando hay un problema con la solicitud (ej. URL inválida o fallo de red).
- `progress` → Se activa durante la descarga de la respuesta, informando el progreso.

```
xhr.addEventListener("error", function () {  
    alert(`Error de red`);  
});
```

7. Enviar la petición al servidor

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

- `body` (opcional): Contiene el cuerpo de la solicitud, usado en métodos como `"POST"`.
- **Valores válidos:** `string`, `FormData`, `URLSearchParams`, etc.
- **Si el método es `"GET"`**, el parámetro `body` se ignora y el cuerpo de la petición será `null`.

Enviar datos al servidor

1. Llamada sin parámetros (GET o POST)

Se puede realizar una solicitud GET o POST sin parámetros.

Cliente (JavaScript)

```
xhr.open("GET", "./hola.txt");  
xhr.send();
```

Servidor (.txt, .json, .php)

Puede ser un archivo de texto, JSON o PHP que devuelva contenido mediante `echo`.

2. Llamada con parámetros no JSON (GET)

Se envían parámetros en la URL.

Cliente (JavaScript)

```
let url = "./buscar.php?nombre=" + campoNom.value + "&apellido=" + campoAp
xhr.open("GET", url);
xhr.send();
```

Servidor (PHP)

```
<?php
    $nombre = $_GET["nombre"];
    $apellido = $_GET["apellido"];
    // ...
?>
```

3. Llamada con parámetros JSON (GET)

Se envían datos en formato JSON codificado en la URL.

Cliente (JavaScript)

```
let user = {
    nombre: document.getElementsByName("name")[0].value,
    apellido: document.getElementsByName("surname")[0].value,
};
xhr.open("GET", "archivo.php&user=" + JSON.stringify(user));
xhr.send();
```

Servidor (PHP)

```
<?php
    $json = $_GET["user"];
    $data = json_decode($json);
    $nombre = $data->nombre;
    $apellido = $data->apellido;
    // ...
?>
```

4. Llamada con parámetros no JSON (POST)

Se envían datos en el cuerpo de la solicitud con `application/x-www-form-urlencoded`.

Cliente (JavaScript)

```
let url = "./buscar.php";
xhr.open("POST", url);
xhr.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
xhr.send("nombre=" + campoNom.value + "&apellido=" + campoApe.value);
```

Servidor (PHP)

```
<?php
    $nombre = $_POST["nombre"];
```

```
$apellido = $_POST["apellido"];
// ...
?>
```

5. Llamada con `FormData` desde un formulario (POST)

Se envían los datos de un formulario sin necesidad de definir cabeceras.

Cliente (HTML)

```
<form id="formElem">
  <input type="text" name="nombre" value="Juan" />
  <input type="text" name="apellido" value="Sierra" />
  <input type="submit" />
</form>
```

Cliente (JavaScript)

```
let formData = new FormData(document.getElementById("formElem"));

xhr.open("POST", "usuario.php");
xhr.send(formData);
```

Servidor (PHP)

```
<?php
  $nombre = $_POST["nombre"];
  $apellido = $_POST["apellido"];
  // ...
?>
```

6. Llamada con `FormData` desde variables (POST)

En este caso no es necesario enviar cabeceras.

Cliente (JavaScript)

```
let formData = new FormData();
formData.append("nombre", "Juan");
formData.append("apellido", "Sierra");

xhr.open("POST", "usuario.php");
xhr.send(formData);
```

Servidor (PHP)

```
<?php
  $nombre = $_GET["nombre"];
  $apellido = $_GET["apellido"];
```

```
// ...  
?>
```

7. Llamada con datos en formato JSON (POST)

Se envía un objeto JSON en el cuerpo de la petición.

Cliente (JavaScript)

```
let user = {  
  nombre: document.getElementsByName("name")[0].value,  
  apellido: document.getElementsByName("surname")[0].value,  
};  
  
xhr.open("POST", "archivo.php");  
xhr.setRequestHeader("Content-Type", "application/json; charset=utf-8");  
xhr.send(JSON.stringify(user));
```

Servidor (PHP)

```
<?php  
$json = file_get_contents('php://input');  
$data = json_decode($json);  
  
$nombre = $_GET["nombre"];  
$apellido = $_GET["apellido"];  
// ...  
?>
```

Recibir datos del servidor

1. Respuesta en texto

Se obtiene la respuesta como texto.

Cliente (JavaScript)

```
xhr.addEventListener("load", function () {  
  if (xhr.status !== 200) {  
    alert(`Error ${xhr.status}: ${xhr.statusText}`);  
  } else {  
    let texto = xhr.response;  
    // tratar el texto;  
  }  
});
```

Servidor (.txt, .json, .php)

Puede devolver una respuesta en texto o JSON con `echo`.

2. Respuesta en JSON

El servidor devuelve una respuesta JSON, que se decodifica automáticamente en JavaScript.

Cliente (JavaScript)

```
xhr.responseType = "json";
xhr.addEventListener("load", function () {
    if (xhr.status !== 200) {
        alert(`Error ${xhr.status}: ${xhr.statusText}`);
    } else {
        objeto = xhr.response;
    }
});
```

Servidor (PHP)

```
<?php
//Creamos la consulta SQL
$sql = "SELECT idAlumno, alumno, puntuacion FROM $objeto->tabla
        WHERE puntuacion >= $objeto->valor";

//Ejecutamos la consulta
$resultado = $conexion->query($sql);

//Almacenamos el resultado en un array asociativo
$salida = array();
$salida = $resultado->fetch_all(MYSQLI_ASSOC);

//Codificamos el array a JSON
echo json_encode($salida);
?>
```

La API URL

La API `URL` facilita la manipulación de URLs, permitiendo obtener y modificar sus componentes como el protocolo, el dominio, el puerto, los parámetros de búsqueda y más. También permite construir nuevas URLs o modificar las existentes de manera sencilla.

Crear una URL

```
let url = new URL(url, [base]);
```

- `url` → Cadena que representa la URL absoluta o relativa.
- `base` (opcional) → Base de referencia si `url` es relativa.

Se puede utilizar como base un objeto URL. En ese caso, se toma la propiedad `href`.

Si `url` es relativa, el parámetro `base` es obligatorio para que se resuelva correctamente. También se puede usar un objeto `URL` como base, utilizando su propiedad `.href`.

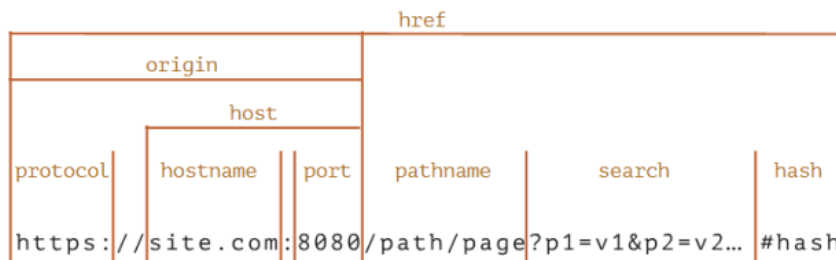
```
let url = new URL("hola.txt", window.location.href);
```

```
console.log(url.href); // URL completa basada en la ubicación actual
```

Propiedades del objeto URL

Algunas de las propiedades más utilizadas son:

- `href` → Devuelve la URL completa (equivalente a `url.toString()`).
- `protocol` → Protocolo de la URL (ej. `"https:"`).
- `host` → Nombre del dominio y puerto (ej. `"www.ejemplo.com:8080"`).
- `hostname` → Nombre del dominio sin el puerto.
- `pathname` → Ruta del recurso dentro del servidor (ej. `"/ruta/pagina.html"`).
- `search` → Cadena de parámetros de búsqueda, incluyendo `?` (ej. `"?q=javascript"`).
- `hash` → Fragmento de la URL después de `#`.
- `username` y `password` → Datos de autenticación si están incluidos en la URL (ej. `http://usuario:contraseña@ejemplo.com`).



Manipulación de parámetros de búsqueda con `URLSearchParams`

La interfaz `URLSearchParams` permite gestionar los parámetros de búsqueda (`query parameters`) de una URL de forma sencilla.

```
let url = new URL("https://www.google.com");
url.searchParams.set("q", "javascript");
console.log(url.href);
// 'https://www.google.com/?q=javascript'
```

Si el parámetro contiene espacios o caracteres especiales, se codifican automáticamente:

```
url.searchParams.set("q", "javascript info");
console.log(url.href);
// 'https://www.google.com/?q=javascript+info'
```

Podemos crear un objeto `URLSearchParams` para enviar datos en una solicitud POST con `application/x-www-form-urlencoded`:

```
xhr.open("POST", url);
xhr.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");

let params = new URLSearchParams();
```



```
params.set("localidad", campolocalidad.value);

xhr.send(params);
```

Promesas

Las promesas son objetos introducidos en ES6 (ES2015) que permiten gestionar operaciones asíncronas de manera eficiente. Se utilizan principalmente para manejar llamadas a servidores y evitar bloqueos en la ejecución del código.

Las promesas actúan como intermediarios entre el código del cliente y el servidor, permitiendo procesar resultados exitosos o errores sin necesidad de recargar la página.

Creación de una promesa

Una promesa se crea con el constructor `Promise`:

```
let promise = new Promise(function(resolve, reject) {
    // Código asíncronico (productor)
});
```

- `resolve(valor)` → Se llama cuando la operación se completa con éxito.
- `reject(error)` → Se llama cuando ocurre un error.
- **Código productor:** Se ejecuta inmediatamente al crear la promesa.

```
// Promesa resuelta
let promise = new Promise(function(resolve, reject) {
    setTimeout(() => resolve("Completado"), 1000);
});

// Promesa rechazada
let promise = new Promise(function(resolve, reject) {
    setTimeout(() => reject(new Error("Ocurrió un error")), 1000);
});
```

Estados de una promesa

Las promesas tienen dos propiedades internas:

- **state (estado):**
 - `pending` → Inicial, en espera.
 - `fulfilled` → Resuelta con éxito (`resolve`).
 - `rejected` → Rechazada por error (`reject`).
- **result (resultado):**
 - `undefined` → Estado inicial.
 - `valor` → Si se llama a `resolve(valor)`.

- `error` → Si se llama a `reject(error)`.

Importante:

- Una promesa ejecuta su código inmediatamente.
- Solo puede tener un resultado: éxito o error.
- Llamadas adicionales a `resolve()` o `reject()` son ignoradas.

Métodos `then`, `catch` y `finally`

Las promesas tienen tres métodos principales para gestionar su resultado.

1. `.then(funcionOk, funcionError)`: Ejecuta una función si la promesa se resuelve (`resolve`), y opcionalmente otra si se rechaza (`reject`).

```
promise.then(  
  result => console.log(result), // Se ejecuta si la promesa se resuelve  
  error => console.error(error)  // Se ejecuta si la promesa se rechaza  
);
```

```
new Promise(resolve => setTimeout(() => resolve("Éxito"), 1000))  
  .then(result => console.log(result)); // "Éxito"
```

2. `.catch(funcionError)`: Captura errores cuando la promesa es rechazada.

```
promise.catch(error => console.error(error));  
  
// Equivalente a:  
promise.then(null, error => console.error(error));
```

```
new Promise((resolve, reject) => setTimeout(() => reject("Error"), 1000))  
  .catch(error => console.error(error)); // "Error"
```

3. `.finally(funcion)`: Se ejecuta siempre, independientemente del resultado (éxito o error).

```
promise.finally(() => console.log("Finalizado"));
```

- No recibe argumentos.
- Se usa para tareas generales, como limpiar recursos o detener un loader.

Encadenamiento de promesas

El método `.then` devuelve una nueva promesa, permitiendo encadenar varias operaciones.

1. Encadenamiento con valores

```
new Promise(resolve => setTimeout(() => resolve(1), 1000))  
  .then(result => {  
    console.log(result); // 1  
  })
```

```

        return result * 2;
    })
    .then(result => {
        console.log(result); // 2
        return result * 2;
    })
    .then(result => console.log(result)); // 4

```

2. Encadenamiento con promesas

```

new Promise(resolve => setTimeout(() => resolve(1), 1000))
    .then(result => {
        console.log(result); // 1
        return new Promise(resolve => setTimeout(() => resolve(result * 2))
    })
    .then(result => {
        console.log(result); // 2
        return new Promise(resolve => setTimeout(() => resolve(result * 2))
    })
    .then(result => console.log(result)); // 4

```

La API Fetch

Introducción

La API `Fetch` permite realizar solicitudes HTTP de manera asíncrona, proporcionando una alternativa moderna a `XMLHttpRequest` para recuperar recursos de la web de forma más sencilla y eficiente.

Método `fetch()`

```
fetch(recurso, [opciones])
```

- **recurso** (obligatorio): URL del recurso a solicitar.
- **opciones** (opcional): objeto con configuración adicional, como:
 - **method** → `"GET"` (por defecto) o `"POST"`.
 - **headers** → objeto con cabeceras HTTP.
 - **body** → datos a enviar (solo en `"POST"`).
 - **mode** → `"cors"`, `"same-origin"`, etc.

```

// GET - Obtener un archivo
fetch("archivo.txt");

// POST - Enviar datos
fetch("archivo.php", {
    method: "POST",

```

```
body: "param=valor",
headers: { "Content-type": "application/x-www-form-urlencoded" },
});
```

Objeto `Response`

Al recibir una respuesta de `fetch()`, se obtiene un objeto `Response` con estas propiedades:

Propiedad	Descripción
<code>.headers</code>	Cabeceras de la respuesta.
<code>.ok</code>	<code>true</code> si el estado es 200-299, <code>false</code> en otros casos.
<code>.status</code>	Código de estado (ej. <code>200</code>).
<code>.statusText</code>	Mensaje asociado al código (<code>OK</code> , <code>Not Found</code>).
<code>.type</code>	Tipo de respuesta (<code>cors</code> , <code>same-origin</code>).
<code>.url</code>	URL del recurso solicitado.

Métodos de `Response` :

Método	Retorno
<code>.json()</code>	Promesa con un objeto JSON.
<code>.text()</code>	Promesa con el contenido como texto.

Manejo de respuestas con `fetch()`

1. **Realizar la solicitud con `fetch()`** : `fetch()` devuelve una promesa que se resuelve con un objeto de tipo `Response` .
2. **Verificar si la respuesta fue exitosa (`response.ok`)**: Si `response.ok` es `false` , se lanza un error.
3. **Procesar los datos de la respuesta**: Se usa `.json()` o `.text()` para extraer la información según el formato esperado.

```
fetch(url, opciones)
  .then(response => {
    if (!response.ok) {
      throw new Error(`Error HTTP: ${response.status}`);
    }
    return response.json();
  })
  .then(datos => {
    tratar(datos);
  })
  .catch(error => console.error("Error en la petición:", error));
```

Uso de `async` y `await`

Las palabras clave `async` y `await` simplifican el manejo de promesas y facilitan la lectura del código.

- `async` → Indica que una función devuelve una promesa.
- `await` → Pausa la ejecución hasta que la promesa se resuelva.

```
async function cargar(url) {
  try {
    let response = await fetch(url);
    if (!response.ok) throw new Error(`Error HTTP: ${response.status}`);

    let datos = await response.json();
    tratar(datos);
  } catch (error) {
    console.error("Error en la carga:", error);
  }
}

cargar("./archivo.json");
```

1. GET - Obtener un archivo de texto

```
async function cargarTexto(url) {
  try {
    let response = await fetch(url);
    if (!response.ok) throw new Error(response.statusText);

    let contenido = await response.text();
    console.log(contenido);
  } catch (error) {
    console.error("Error:", error);
  }
}

cargarTexto("./hola.txt");
```

2. GET - Obtener un JSON

```
async function cargarJSON(url) {
  try {
    let response = await fetch(url);
    if (!response.ok) throw new Error(response.statusText);

    let datos = await response.json();
    console.log(datos);
  } catch (error) {
    console.error("Error:", error);
  }
}
```

```
cargarJSON("./datos.json");
```

3. POST - Enviar datos como `x-www-form-urlencoded`

```
async function enviarDatos(url, nombre, apellido) {
  let opciones = {
    method: "POST",
    body: `nombre=${encodeURIComponent(nombre)}&apellido=${encodeURIComponent(apellido)}`,
    headers: { "Content-Type": "application/x-www-form-urlencoded" },
  };

  try {
    let response = await fetch(url, opciones);
    if (!response.ok) throw new Error(response.statusText);

    let resultado = await response.text();
    console.log(resultado);
  } catch (error) {
    console.error("Error:", error);
  }
}

enviarDatos("./procesar.php", "Juan", "Pérez");
```

4. POST - Enviar un objeto JSON

```
async function enviarJSON(url, datos) {
  let opciones = {
    method: "POST",
    body: JSON.stringify(datos),
    headers: { "Content-Type": "application/json; charset=UTF-8" },
  };

  try {
    let response = await fetch(url, opciones);
    if (!response.ok) throw new Error(response.statusText);

    let resultado = await response.json();
    console.log(resultado);
  } catch (error) {
    console.error("Error:", error);
  }
}

enviarJSON("./procesar.php", { nombre: "Ana", edad: 25 });
```

5. POST - Enviar un `FormData`

```

async function enviarFormulario(url, localidad) {
    let formData = new FormData();
    formData.append("localidad", localidad);

    let opciones = {
        method: "POST",
        body: formData,
    };

    try {
        let response = await fetch(url, opciones);
        if (!response.ok) throw new Error(response.statusText);

        let resultado = await response.text();
        console.log(resultado);
    } catch (error) {
        console.error("Error:", error);
    }
}

enviarFormulario("./procesar.php", "Madrid");

```