

PROCESO DE GENERACIÓN DE APLICACIONES DINÁMICAS MODERNAS

AJAX significa Asynchronous JavaScript And XML, y es una técnica que permite que el navegador se comunique con un servidor sin recargar la página completa.

No es una tecnología por sí misma, sino un concepto / técnica de desarrollo web. El xml es el formato con el que se enviaban y recibían los datos, ahora normalmente se usa JSON.

Antes de AJAX, cada vez que querías obtener datos del servidor (por ejemplo, una lista de productos filtrados), se tenía que recargar toda la página.

Esto generaba experiencias lentas y rompía la interacción del usuario.

Cómo funciona:

1. Se hace una petición HTTP desde JavaScript.
2. El servidor procesa la petición y devuelve datos (JSON, XML, HTML...).
3. JS recibe esos datos y actualiza solo la parte de la página que corresponde.

Es decir, permite construir aplicaciones web interactivas tipo SPA (Single Page Application) sin recargar la página.

La tecnología que se usaba para implementar AJAX en su día era XMLHttpRequest, API antigua y un poco engorrosa.

La tecnología utilizada actualmente, **fetch**, resuelve esto:

- Sintaxis más limpia y legible.
- Basado en promesas, lo que facilita encadenar operaciones (.then() y .catch()).

Una promesa es un objeto que representa un valor que estará disponible en el futuro. Algo así como: "Prometo que cuando la petición termine, te daré el resultado o te diré que falló".

- Estados de una promesa:
 1. pending (pendiente) → aún no se ha completado ni fallado.
 2. fulfilled (resuelta) → la operación terminó correctamente.
 3. rejected (rechazada) → ocurrió un error.
- Métodos de las promesas:
 - .then(callback) → se ejecuta cuando la promesa se resuelve. Es el único obligatorio.
 - .catch(callback) → se ejecuta si la promesa falla. Si queremos capturar un error es útil.
 - .finally(callback) → se ejecuta siempre al final, haya sido resuelta o rechazada.

¿Qué conseguimos mejorar con esto?

- **Antes, en desarrollo clásico:**
 - Cada interacción con el servidor requería recargar la página.
 - Las experiencias eran lentas y poco interactivas.
 - HTML y lógica de servidor estaban muy acoplados.
- **Con Fetch + Promesas:**
 - Se actualiza solo la parte de la página necesaria.
 - La interfaz de usuario permanece fluida y no se pierde el estado (scroll, inputs, filtros...).
 - **Se separa claramente:**
 - PHP o servidor: maneja la lógica y devuelve datos.
 - JavaScript: actualiza la presentación.
 - Se pueden encadenar múltiples operaciones asíncronas fácilmente.
 - En esto se basan los frameworks y tecnologías de desarrollo modernos

¿Y cómo manda fetch los datos al servidor?

A través de FormData.

FormData es un objeto nativo de JavaScript que permite estructurar los campos de un formulario HTML para enviarlos a un servidor mediante fetch.

Esencialmente: toma todos los <input>, u otros elementos en los que mandamos información desde un formulario y empaqueta su nombre y valor en un formato que el servidor puede entender (multipart/form-data).

Ejemplo de fetch + FormData para cuando ocurre evento submit en formulario de búsqueda:

```
document.getElementById("filtro").addEventListener("submit", e => {
  .preventDefault(); -> Primero prohibimos que cuando hagamos submit se recargue toda la página
  fetch("buscador.php", {
    method: "POST",
    body: new FormData(e.target)
  })
})
```

Lo que significa este código, es que fetch va a hacer una petición POST, la cual va a procesar el script PHP llamado buscador.php, mandando los datos en un objeto FormData.

Para que funcione, e.target debe ser evidentemente un formulario.

¿Cuándo recibe la petición PHP qué hace?

Algo muy parecido a lo que hacíamos siempre con la visión clásica. Recogemos los valores del formulario y ejecutamos la sql de búsqueda:

```
$texto = $_POST["busqueda"];
$sql = "SELECT nombre, precio FROM productos WHERE nombre LIKE '%$texto%'";
$result = $conexion->query($sql);
```

PERO SEGUIDAMENTE NO MOSTRAMOS NINGÚN RESULTADO CON ECHOS NI PRODUCIENDO HTML, VAMOS A GUARDAR EN UN ARRAY TODO LO EXTRAÍDO DE BASE DE DATOS:

```
$salida = []; // array vacío
if ($resultado->num_rows > 0) {
    while ($row = $resultado->fetch_assoc()) {
        $salida[] = $row; // cada fila se añade al array
    }
}
```

INDICAMOS AL CLIENTE QUE LO QUE ENVIAREMOS SERÁ EN FORMATO JSON:

```
header("Content-Type: application/json");
```

Y POR ÚLTIMO CONVERTIMOS EL ARRAY A JSON Y LO ENVIAMOS AL CLIENTE

```
echo json_encode($salida);
```

¿Una vez hemos recibido en el cliente los datos que ha mandado el servidor en ese formato JSON, qué se hace en JavaScript?

```
.then(r => r.json())
```

Es la forma abreviada de escribir:

```
.then(function(r) {
    return r.json();
});
```

Este fragmento, al estar dentro de un .then se ejecuta cuando fetch() recibe una respuesta del servidor. Es decir, cuando la promesa se ha resuelto.

r es el nombre que le damos al parámetro que recibimos en la función

Representa el objeto Response (respuesta HTTP) que nos devuelve fetch.

Ese objeto tiene información como:

- r.status → código HTTP (200, 404, 500...)

- r.ok → true/false si la petición fue correcta
- r.headers → cabeceras enviadas por PHP
- r.body → el contenido REAL devuelto (nuestros datos JSON)

r.json() es un método del objeto Response, que lee el, lee el cuerpo (body) de la respuesta, que está en forma de *texto* (JSON), y convierte ese texto a un objeto JavaScript.

Una vez hecho esto, con otro .then (si ponemos varios seguidos hasta que no se resuelva el anterior no se ejecuta este), se toma lo que devolvió r.json y se construye en una variable la traducción a html de la información que se devolvió:

```
.then(lista => {
  let html = "";
  lista.forEach(p =>
    // Creamos un <li> para cada producto con nombre y precio
    html = html + `<li>${p.nombre} - ${p.precio} Euros</li>`;
  );
}
```

Ahora ya tenemos todo preparado para mostrar por pantalla únicamente lo devuelto por el servidor, sin recargar la página entera, y con todas las ventajas mencionadas:

Nota adicional:

Si os fijáis, os ha podido sorprender que para una búsqueda de productos hayamos usado POST. Esto es así porque con esta forma de desarrollar las ventajas clásicas de GET desaparecen, porque:

- No vas a guardar esta URL como favorito, La página siempre es la misma: index.html
- No vas a navegar entre páginas distintas, todo ocurre en la misma.
- No hay recarga completa, por lo tanto la URL deja de ser el estado de la búsqueda.
- Además, ciertos aspectos técnicos como usar FormData solo funcionan con POST.