UNIDAD 6. Mecanismos de Comunicación Asíncrona: AJAX

Desarrollo Web en Entorno Cliente

2º DAW

Contenidos

Introducción

AJAX es un acrónimo que significa *Asynchronous JavaScript And XML* (JavaScript Asíncrono y XML) y es una técnica para el desarrollo de aplicaciones web interactivas o RIA (*Rich Internet Applications*),

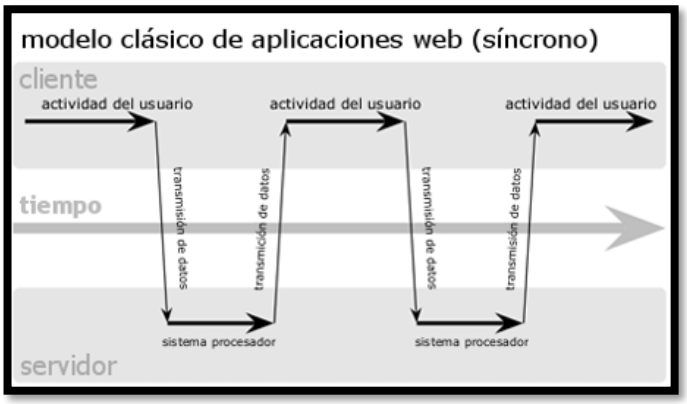
AJAX no es un lenguaje de programación, es el uso combinado de otras tecnologías existentes:

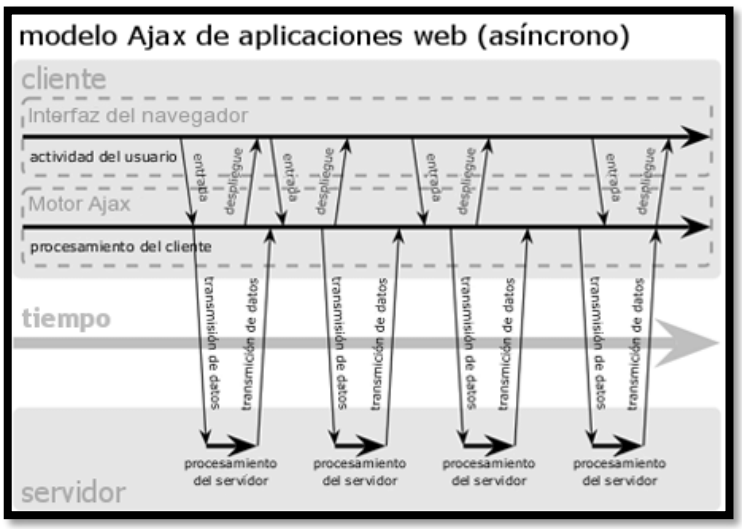
- Objeto `XMLHttpRequest` integrado en el navegador: para solicitar datos de un servidor web.
- JavaScript
- HTML DOM: para mostrar o interactuar con los datos.
- XML u otro formato a través del que sea posible la transferencia de datos solicitados al servidor, como archivos de texto sin formato, JSON, PHP, etc.

AJAX permite que las páginas web se actualicen de forma asíncrona mediante el intercambio de datos con un servidor web en segundo plano. Esto significa que es posible actualizar partes de una página web, sin recargar toda la página.

Las peticiones de datos que se realizan a través de AJAX al servidor se conocen como **peticiones asíncronas**, porque se hace la petición al servidor y se sigue ejecutando el código en el cliente, es decir, no se espera a que el servidor conteste, **mejorando la interactividad, velocidad y usabilidad en las aplicaciones**.

El servidor modifica unas propiedades para indicar su estado; y existen eventos asociados al cambio de estado, que el cliente consulta para saber si el servidor ha terminado.





Se dice que AJAX es el sueño de todo desarrollador web, porque puede:

- Leer datos de un servidor web, después de que se haya cargado la página.
- Actualizar una página web sin recargar la página.
- Enviar datos a un servidor web en segundo plano.

Para trabajar en esta unidad, necesitamos simular en local el trabajo cliente-servidor, por lo que tendremos que abrir los ejercicios en un servidor, por ejemplo: Apache, a través de XAMPP. También tenemos la opción de subir nuestros ejercicios a algún alojamiento web, pero nos resultará más sencillo y rápido trabajar con XAMPP.

El objeto XMLHttpRequest

Introducción

El objeto **`XMLHttpRequest` (XHR)** es un objeto JavaScript que proporciona una forma fácil de obtener información de una URL sin tener que recargar la página completa. Lo podremos utilizar para actualizar una parte de la página web sin interrumpir lo que el usuario está haciendo, por lo tanto, es ampliamente usado en la programación AJAX.

A pesar de lo que puede parecer, `XMLHttpRequest` se puede utilizar para recuperar cualquier tipo de datos, no solo XML, y soporta otros protocolos, además de HTTP (ej. file o ftp).

Por razones de seguridad, los navegadores modernos no permiten el acceso a través de dominios. Esto significa que tanto la página web como el archivo de datos que intenta cargar, deben estar ubicados en el mismo servidor, ya que se sigue la política de seguridad **same-origin-policy** que ya conocemos.

Las últimas versiones de los navegadores, pueden utilizar la API `Fetch` en lugar del objeto `XMLHttpRequest`. Esta interfaz, permite al navegador web realizar solicitudes HTTP a los servidores web, es decir, puede hacer lo mismo que `XMLHttpRequest` pero de una manera más simple.

Propiedades de `XMLHttpRequest`

Algunas propiedades de `XMLHttpRequest`:

Contenido de la respuesta	Estado de la respuesta	Datos de la respuesta	Manejador de eventos
`responseText`	`readyState`	`responseURL`	`readystatechange`
`responseXML`	`status`	`responseType`	
	`statusText`		

Todas las propiedades son de **solo lectura**, excepto `readystatechange`.

<table>
<tr><th>Propiedad</th><th>Contenido</th></tr>
<tr><td><code>.responseText</code></td><td>Devuelve la respuesta del servidor, en formato cadena de texto, o <code>null</code> si la petición no ha tenido éxito o no ha sido enviada todavía.</td></tr>
<tr><td><code>.responseXML</code></td><td>Devuelve un objeto <code>Document</code> que contiene la respuesta de la petición, o <code>null</code> si no ha tenido éxito o si todavía no ha sido enviado, o no puede ser convertida a XML o HTTP.</td></tr>
<tr><td><code>.readyState</code></td><td>Devuelve un entero corto, que indica el estado de la petición. Los posibles valores son:
<table>
<tr><th>Valor</th><th>Estado</th><th>Descripción</th></tr>
<tr><td>0</td><td>UNINITIALIZED</td><td>El cliente XMLHttpRequest se ha creado, pero aún no se ha llamado al método <code>open()</code>.</td></tr>
</table>
</td></tr>
</table>

	1	LOADING	Conexión establecida. Se ha llamado al método `open()`, pero aún no se ha llamado al método `send()`.
	2	LOADED	Se ha recibido la petición, lo que siginifica que se ha llamado a send() y la cabecera HTTP está disponible
	3	INTERACTIVE	Se está procesando la respuesta, es decir, descargando los datos del servidor.
	4	COMPLETED	Operación completa: petición finalizada y respuesta completada.
`.status`	Devuelve un entero corto con el código de respuesta HTTP. Antes de que la petición termine, el valor de status será 0. Se utilizan los códigos estándar, por ejemplo: o 200: OK o 403: Forbidden o 404: Not Found		
`.statusText`	Devuelve una cadena correspondiente a la respuesta del servidor. A diferencia de `.status`, incluye el texto completo del mensaje (ej. "200 OK").		
`.readystatechange`	Evento que ocurre cuando cambia el valor de la propiedad `readyState`.		

Métodos de `XMLHttpRequest`

Algunos métodos de `XMLHttpRequest`:

Método	Devuelve
`new XMLHttpRequest()`	Método constructor, crea un nuevo objeto `XMLHttpRequest`.
`.abort()`	Cancela la petición, si ha sido enviada.
`.getAllResponseHeaders()`	Devuelve una cadena con todas las cabeceras de la respuesta, separadas por salto de línea. Si no se ha recibido respuesta, devuelve `null`.
`.getResponseHeader(`*`cab`*`)`	Devuelve una cadena con el texto de una cabecera específica indicada como argumento. Si no se ha recibido respuesta, devuelve `null`.
`.open(`*`method, url, [async, user, password]`*`)`	Inicializa una petición. Parámetros: • `method`: método utilizado en la petición HTTP. Puede ser GET o POST. Se ignora en URLs que no sean HTTP. • `url`: URL el archivo sobre el que se hace la petición, puede ser un archivo .txt, .php, .xml, .json, etc. • `async` (opcional, por defecto `true`) indica si la petición se hará de modo asíncrono o no (`false`): • `user` (opcional, por defecto vacío): nombre del usuario. • `password` (opcional, por defecto vacío): contraseña del usuario.
`.send(`*`[body]`*`)`	Envía la petición al servidor. Si la petición es asíncrona (como ocurre por defecto), este método termina en cuanto

	envía la petición, en caso contrario, no termina hasta que llegue la respuesta. Tiene un parámetro opcional `body` (por defecto `null`) con los datos que se quieren enviar al servidor. Ejemplo, en una petición POST: `xhr.send("nombre=pepe&estudios=DAW");`
`.setRequestHeader()`	Establece el valor de una cabecera de petición HTTP. Se debe llamar a este método después de `open()`, pero antes de `send()`. Ejemplo, para una petición POST: `xhr.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");`

Pasos a seguir para trabajar con `XMLHttpRequest`

Los siguientes pasos se pueden aplicar en general para trabajar con cualquier tipo de archivo alojado en el servidor, pero iremos viendo ejemplos con los más habituales, ya que tienen algunas particularidades:

1. **Crear el objeto `XMLHttpRequest`**
2. **Asignar el manejador de eventos**: definimos la función que va a ser invocada cuando cambie la propiedad `readyState`. Dentro de esa función:
   - Comprobar que tenemos el valor 4 en la propiedad `readyState` (petición al servidor finalizada y respuesta completada) y el valor 200 en la propiedad `status` (petición OK).
   - Trabajar con la propiedad `responseText` o con `responseXML` según sea el tipo de archivo alojado en el servidor: .txt, .php, .xml, .json, etc.
3. **Inicializar el objeto `XMLHttpRequest`** es decir, definir la conexión con el método `open("GET|POST", "archivo")`
   - GET: más rápido, límite de caracteres de 2000 y el envío se realiza a través de URL, por lo tanto, no es útil para grandes cantidades de datos, ni para contraseñas u otros datos sensibles.
   - POST: es la opción más usada, ya que encripta los parámetros, lo que aporta seguridad.
4. **Enviar la petición al servidor** con `send()`. Si hemos utilizado POST en el paso anterior, debemos pasar los datos como parámetro y haber enviado las cabeceras con `setRequestHeader()`.

API `Fetch` y promesas

Introducción

La **API `Fetch`** nos permite trabajar con peticiones y respuestas HTTP, además, nos aporta una forma fácil y lógica de obtener recursos de forma asíncrona por la red, es decir, de realizar transacciones AJAX.

La API `Fetch` de ES6, permite al navegador web realizar solicitudes HTTP a los servidores web, es decir, puede hacer lo mismo que `XMLHttpRequest` pero de una manera más simple,

especialmente que desde que se incorporaron en ES8 las palabras clave `async` y `await`, que facilitan mucho el uso de promesas.

Al tratarse de opciones incorporadas de forma tan reciente, los navegadores aún no las tienen estandarizadas, pero debemos aprender su uso, ya que han recibido tan buena aceptación, que son el futuro de AJAX.

Tampoco hay que hay que olvidar que el objeto `XMLHttpRequest` sigue siendo interesante, ya que permite hacer algunas cosas que este nuevo API no es capaz de hacer, como, por ejemplo, llevar un control del progreso de la transferencia de datos entre el navegador y el servidor web, o la capacidad de cancelar una petición AJAX.

Método `fetch()`

Sintaxis:

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

Argumentos:

- `recurso`: obligatorio, cadena que contiene la ruta de acceso al recurso que desea recuperar.
- `opciones`: opcional, objeto que representa un conjunto de opciones de configuración para personalizar la solicitud HTTP. Algunas de esas opciones:
  - `method`: método de solicitud, GET (por defecto) o POST.
  - `headers`: cualquier cabecera que se quiera añadir a la solicitud, contenidas en un objeto `Headers` o un objeto literal.
  - `body`: cualquier cuerpo que se quiera añadir a la solicitud, las solicitudes con métodos GET no pueden tener cuerpo.
  - `mode`: el modo de la solicitud: cors, same-origin.

Ejemplo: transacción GET

```
let url = "archivo.txt";
fetch(url)
```

Ejemplo: transacción POST

```
let url = "archivo.php";
let opciones = {
    method: "POST", //GET o POST
    body: "param = valor", //string u {object} solo con POST
    headers: {
        "Content-type": "application/x-www-form-urlencoded"
    } //Pares clave:valor para las cabeceras
};
fetch(url, opciones)
```

El método `fetch()` lanza el proceso de solicitud de un recurso de la red y devuelve una **promesa** con un objeto `Response`. Una vez hemos obtenido el objeto `Response`, hay varios métodos disponibles para definir cuál es el contenido del cuerpo y como se debe manejar.

Que un método devuelva una promesa, implica que, a la hora de utilizarlos, tenemos que tratarlos de una forma especial, como veremos más adelante.

Propiedades de `Response`

Algunas propiedades del objeto `Response`, todas de solo lectura:

Propiedad	Contenido
`.headers`	Contiene el objeto `Headers` asociado a la respuesta.
`.ok`	Su valor es `true` si la respuesta tuvo éxito (rango 200-299) o `false` en caso contrario.
`.status`	Contiene el código de estado de la respuesta (por ejemplo: 200 si tuvo éxito).
`.statusText`	Contiene el mensaje de estado correspondiente al código de estado (por ejemplo: OK para el código 200).
`.type`	Contiene el tipo de respuesta (por ejemplo: same-origin, cors).
`.url`	Contiene la URL de respuesta.

Métodos de `Response`

Algunos métodos de `Response`:

Método	Devuelve
`.arrayBuffer()`	Devuelve una promesa un objeto `ArrayBuffer`, que representa un buffer genérico de datos binarios.
`.blob()`	Devuelve una promesa con un objeto `Blob`, que representa un objeto tipo fichero de datos planos inmutables, como, por ejemplo, una imagen.
`.formData()`	Devuelve una promesa con un objeto `FormData`, que proporciona una manera fácil de construir un conjunto de pares clave / valor que representan campos de un formulario y sus valores.
`.json()`	Devuelve una promesa con un objeto JSON.
`.text()`	Devuelve una promesa con un objeto que almacena texto plano.

Promesas, `async` y `await`

Una **promesa**, es un objeto de la clase **`Promise`** de ES6 que se utiliza en operaciones asíncronas. Representan tareas que devuelven un valor que puede estar disponible ahora, en el futuro, o nunca.

Una promesa tiene tres estados posibles:

- pendiente: antes de ejecutar la tarea asociada, devuelve `undefined`.
- cumplida: la tarea tiene éxito y devuelve el valor prometido.
- rechazada: la tarea falla y devuelve lanza una excepción con un código de error.

Las promesas ordenan la ejecución de funciones (*callbacks*) en el tiempo, y conservan la eficiencia de ejecución de los *callbacks* asíncronos, ya que ceden el procesador a otras funciones

hasta que se resuelven las de la promesa. Se llaman *callbacks* a las funciones que tienen como argumento otra función.

Las palabras clave **`async`** y **`await`** de ES8 facilitan el uso de promesas. Su uso es compatible con las promesas de ES6 y puede mezclarse con ellas. Nosotros vamos a trabajar solamente con esta sintaxis.

La palabra **`async`** se antepone a la definición de la función para conseguir que devuelva una promesa. La promesa finaliza con éxito devolviendo el valor que se especifique en la definición de la función, o es rechazada y lanza una excepción.

Ejemplo:

```
async function suma (x, y) {....}
//O con funciones flecha
async (x, y) => {....}
```

Una función `async` puede contener una expresión **`await`**, que pausa la ejecución de la función asíncrona y espera la resolución de la promesa pasada y, a continuación, reanuda la ejecución de la función `async` y devuelve el valor resuelto.

La palabra `await` se antepone a la llamada a una función que devuelve una promesa y solo se puede utilizar dentro de una función `async`, no se puede utilizar en otros contextos.

Ejemplo: asigna a x el valor devuelto en caso de éxito, en caso de rechazo, lanza una excepción que puede capturarse con una sentencia `try..catch`.

```
Async function y () {
    let x = await promesa()
}
```

Pasos a seguir para trabajar con la API `Fetch` y promesas

Los siguientes pasos se pueden aplicar en general para trabajar con cualquier tipo de archivo alojado en el servidor, pero iremos viendo ejemplos con los más habituales, ya que tienen algunas particularidades:

1. Crear una función asíncrona **`async`** que devuelve una promesa y tendrá como argumento el recurso que queremos obtener del servidor. Dentro de esa función:
    - Utilizar `await` para invocar al método `fetch(url)` que hace la petición del recurso.
    - Trabajar con `await` para invocar al método de la clase `Response` que necesitemos para obtener el contenido del recurso solicitado, por ejemplo: `await objeto.text()`.
    - Una vez obtenido el recurso, lo utilizamos en nuestro script.
2. Invocar a la función asíncrona que hemos creado, pasando como argumento la URL que referencia al recurso que queremos obtener del servidor.

Trabajar con archivos de texto

Para saber cómo trabajar con archivos de texto, vamos a ver un ejemplo completo. En este ejemplo tenemos un documento con un botón y un `<div>`. Al hacer clic sobre el botón **cambiaContenido**, queremos que se cargue en el `<div>` el contenido del archivo de texto **holamundo.txt**, que está en el servidor.

La solución utilizando `XMLHttpRequest`, es la siguiente:

```
<div id="texto">
    <h1>AJAX</h1>
    <button id="cambiaContenido">Cambia el contenido</button>
</div>
```

```
document.getElementById("cambiaContenido").addEventListener("click", cambiaConten
ido);

function cambiaContenido() {
    //1. Crear el objeto XMLHttpRequest
    let xhr = new XMLHttpRequest();
    //2. Asignar el manejador de eventos
    //Dentro uso this, no puede ser función =>
    xhr.addEventListener("readystatechange", function () {
        //Servidor ha terminado con respuesta OK
        if (this.readyState === 4 && this.status === 200) {
            document.getElementById("texto").innerHTML = this
            .responseText; //El fichero está en this.responseText
        }
    });
    //3. Inicializar el objeto XMLHttpRequest
    /* .open: define la conexión
        - GET/POST.
        - Archivo: txt, php, xml, json, etc.
        - true/false: método de envío, por defecto true, asíncrono */
    xhr.open("GET", "holamundo.txt", true);
    // 3.1. En este caso no: SOLO SI POST enviar las cabeceras
    // xhr.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
    //4. Enviar la petición al servidor
    /* .send: envía la solicitud al servidor.
        Si utilizamos POST debemos pasar los datos por parámetro */
    xhr.send();
}
```

Si queremos utilizar la API `Fetch` y promesas, la función `cambiaContenido()` sería del siguiente modo:

```
function cambiaContenido() {
    //1. Crear una función asíncrona: devuelve una promesa
    async function obtenerTexto(url) {
        let objeto = await fetch(url); //usar await para llamar a fetch()
        let texto = await objeto.text(); //usar await y llamar a Response.text()
        document.getElementById("texto").innerHTML = texto;
    }
    //2. Invocar a la función asíncrona con el recurso como argumento
    obtenerTexto("holamundo.txt");
}
```

Trabajar con archivos XML

Trabajar con archivos XML alojados en un servidor web para mostrar sus datos con JavaScript, es muy similar al trabajo con archivos de texto, pero en este caso, la respuesta del servidor se tomará de la propiedad **`responseXML`**.

El contenido de esa propiedad es un objeto de tipo `Document` que se puede recorrer utilizando las métodos del DOM que ya conocemos, como **`getElementsByTagName()`**.

Vamos a ver un ejemplo muy similar al anterior, pero suponiendo que lo que ahora está alojado en el servidor es un archivo XML en vez de un archivo de texto. Al hacer clic sobre el botón **Carga catálogo**, queremos que se cargue en la tabla con `id="demo"` el contenido del documento **catalogo.xml**, que tiene la siguiente estructura:

```
<CD>
    <TITLE>Empire Burlesque</TITLE>
    <ARTIST>Bob Dylan</ARTIST>
    <COUNTRY>USA</COUNTRY>
    <COMPANY>Columbia</COMPANY>
    <PRICE>10.90</PRICE>
    <YEAR>1985</YEAR>
</CD>
```

```
<div id="texto">
    <h1>Colección de CDs</h1>
    <button id="cargaCatalogo">Carga catálogo</button>
</div>
<br>
<table id="demo"></table>
```

```
document.getElementById("cargaCatalogo").addEventListener("click",cargaCatalogo);

function cargaCatalogo() {
    //1. Crear el objeto XMLHttpRequest
    let xhr = new XMLHttpRequest();
    //2. Asignar el manejador de eventos
    xhr.addEventListener("readystatechange", function () {
```

```
        if (this.readyState === 4 && this.status === 200) {
            cargarXML(this);
        }
    });
    //3. Inicializar el objeto XMLHttpRequest
    xhr.open("GET", "catalogo.xml");
    //4. Enviar la petición al servidor
    xhr.send();
}

function cargarXML(xml) {
    let docXML = xml.responseXML;
    let tabla = "<tr><th>Artista</th><th>Titulo</th></tr>";
    //Podemos utilizar la API DOM que conocemos también para XML
    let discos = docXML.getElementsByTagName("CD");
    for (let i = 0; i < discos.length; i++) {
        tabla += "<tr><td>";
        tabla += discos[i].getElementsByTagName("ARTIST")[0].textContent;
        tabla += "</td><td>";
        tabla += discos[i].getElementsByTagName("TITLE")[0].textContent;
        tabla += "</td></tr>";
    }
    document.getElementById("demo").innerHTML = tabla;
}
```

Si queremos trabajar con la `Fetch` tendríamos que obtener el texto plano y parsearlo a XML o bien, parsear el XML a JSON y trabajar con JSON. Esta API no facilita el trabajo con XML, por lo tanto, no mostramos el ejemplo.

Trabajar con archivos PHP

En este caso, suponemos que el archivo que se genera y devuelve el servidor tiene formato de texto, por lo que trabajaremos con la propiedad **`responseText`** de nuevo.

Para comprender el trabajo con archivos PHP a través de AJAX, vamos a ver un ejemplo en el cual tenemos un cuadro de texto y un párrafo que muestra debajo sugerencias. Cuando un usuario teclea, se le van aportando sugerencias de nombres, si es que el servidor las tiene disponibles.

Para ello, se comprueba si hay algo escrito en el `input`, y se envía una petición al servidor. Para controlar cuándo el usuario ha pulsado una letra, se utiliza el evento `keyup`. El contenido a enviar al servidor es el texto escrito hasta el momento, que estará disponible en el `value` del `input`.

Parte del cliente con GET y `XMLHttpRequest`

Con GET, tendremos que especificar las variables en el método `open()`, porque van en la URL:

```
destino.php?variable1=valor1&variable2=valor2&...
```

```
document.addEventListener("DOMContentLoaded", inicio);

function inicio() {
    document.getElementById("nombre").addEventListener("keyup", mostrarNombre);
}

function mostrarNombre(e) {
    let cadena = e.target.value;
    //Otra opción válida:
    //let cadena = document.getElementById("nombre").value;

    if (cadena.length === 0) {//Si al levantar la tecla no hay nada(ej.al borrar)
        document.getElementById("sugerencia").innerHTML = "";
        return;
    } else {
        let xhr = new XMLHttpRequest();
        xhr.addEventListener("readystatechange", function () {
          if (this.readyState === 4 && this.status === 200) {
             document.getElementById("sugerencia").innerHTML = this.responseText;
          }
        });
        xhr.open("GET", "arraynombres.php?nombre=" + cadena);
        xhr.send();
    }
}
```

Parte del servidor

En el servidor, tendremos que recoger el valor de las variables enviadas, por ejemplo utilizando $_REQUEST.

```
$nombre = $_REQUEST["nombre"];
```

Para devolver el archivo, se utiliza el comando `echo cadena`, donde `cadena` es el contenido del archivo.

```
echo ($sugerencia ===""") ? "No hay sugerencias": $sugerencia;
```

Cuando el servidor recibe una petición, lee el contenido de la variable nombre. Compara su contenido con los valores de un array (array de sugerencias) y devuelve una cadena con los nombres que empiezan con el texto pasado como parámetro.

```
<?php
 //Array de nombres
$a = array("Sara","Imanol","Dani","Antonio","David","Igor", "Naroa", "Christian",
 "Joseba", "Angel", "Alex", "Dumitru", "Mikel", "Ivan", "Martin");

//Tomamos el valor del input procedente de la URL
$nombre = $_REQUEST["nombre"];
$sugerencia = "";
```

```php
if ($nombre!=="") {
    $nombre = strtolower($nombre); //Pasamos el nombre a minúsculas
    $long = strlen($nombre);

    foreach($a as $nom) {
    //Cada elemento del array se almacena en $nom en cada iteración
        if (stristr($nombre, substr($nom, 0, $long))) {
        //Si coincide la cadena pasada con los primeros caracteres de algún
        //elemento del array
            if ($sugerencia === "") { //Si no hay texto en sugerencia
                $sugerencia = $nom;
            } else {
                $sugerencia = "$sugerencia, $nom";
            }
        }
    }
}
echo ($sugerencia ===""") ? "No hay sugerencias" : $sugerencia;
?>
```

Parte del cliente con POST y `XMLHttpRequest`

Con POST, tendremos que especificar las variables en el método `send()`, y llamar antes al método `.setRequestHeader()`, es decir, en el ejemplo anterior, tendríamos que cambiar las líneas marcadas en un fondo gris, por las tres sentencias siguientes:

```
//xhr.open("GET", "arraynombres.php?nombre=" + cadena);
xhr.open("POST", "arraynombres.php");
//Con POST tenemos que incluir la cabecera
xhr.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
//Con POST tenemos que pasar los argumentos como parámetros de send, en lugar de
//en la URL
xhr.send("nombre=" + cadena);
```

Parte del cliente con GET y `Fetch`

Dentro de la función `mostrarNombre()` tendríamos que cambiar la parte del `else` por lo siguiente:

```
async function obtenerTexto(url) {
    let objeto = await fetch(url);
    let texto = await objeto.text();
    document.getElementById("sugerencia").innerHTML = texto;
}
obtenerTexto("arraynombres.php?nombre=" + cadena);
```

Parte del cliente con POST y `Fetch`

Dentro de la función `mostrarNombre()` tendríamos que cambiar la parte del `else` por lo siguiente:

```
//Creamos un objeto con configuraciones para personalizar la solicitud
let opciones = {
    method: "POST", //GET o POST
    body: "nombre=" + cadena, //string u {object} solo con POST
    headers: {
        "Content-type": "application/x-www-form-urlencoded"
    } //Objeto de tipo Headers o literal con pares clave:valor para las cabeceras
};
async function obtenerTexto(url) {
    let objeto = await fetch(url, opciones);
    let texto = await objeto.text();
    document.getElementById("sugerencia").innerHTML = texto;
}
obtenerTexto("arraynombres.php");
```

Vemos que solamente se han modificado las líneas marcadas en un fondo gris, en el ejemplo anterior, además de definir el objeto que permite personalizar la configuración de la solicitud `fetch()`.

El formato de intercambio de datos JSON

JSON son las siglas de *JavaScript Object Notation*, que es un formato de serialización de objetos JavaScript.

La **serialización de datos** nos aporta ventajas como:

- Transformación reversible de valores en un string equivalente.
- Facilita el almacenamiento e intercambio de datos, por ejemplo, en los siguientes casos tenemos que trabajar con cadenas:
    - Almacenar o recuperar datos en un fichero alojado en un servidor.
    - Enviar o recibir datos desde o hacia un servidor.
    - Almacenar o recuperar datos con la API `WebStorage`.

Existen otros formatos de serialización como XML, HTML o XDR(C), pero estos formatos están siendo desplazados por JSON, incluso XML, ya que el código resultante es más corto, más rápido de leer y escribir, y además puede usar arrays. Existen bibliotecas de JSON para los lenguajes más importantes http://json.org/json-es.html.

El objeto JSON de JavaScript

Podemos convertir cualquier objeto JavaScript en JSON para enviar o almacenar datos y también podemos recuperar el objeto JavaScript a partir del cual fue creado el JSON con los siguientes métodos:

Método	Devuelve

`JSON.stringify(objeto)`	Devuelve la cadena JSON resultante de transformar un objeto JavaScript en un `string` JSON equivalente.
	Ejemplo:
	`let obj  = { x: 5, y: 6 };`
	`let cadJSON = JSON.stringify(obj);`

`JSON.parse(cadena)`	Devuelve el objeto JavaScript resultante de transformar un `string` JSON en el objeto equivalente.
	Ejemplo:
	`let json = '{"result":true, "count":42}';`
	`let obj = JSON.parse(json);`

Ejemplos:

```
JSON.stringify(null) => 'null'
JSON.parse('null') => null
JSON.stringify(127) => '127'
JSON.stringify('hola') => '"hola"'
JSON.stringify([1, 2, 3]) => '[1, 2, 3]'
JSON.stringify({a:27, b:"hola"}) => '{"a":27,"b":"hola"}'
```

JSON puede serializar:

- objetos, `arrays`, `strings`, números finitos, `true`, `false` y `null`
  - `NaN`, `Infinity` y `-Infinity` se serializan por defecto a `null`
  - Los objetos `Date` se serializan como un `string` en formato ISO 8601 y la reconstrucción devuelve un `string` y no el objeto `Date` original que se serializó

JSON **no** puede serializar:

- Funciones, `RegExp`, errores, `undefined`

Ejemplos:

```
JSON.stringify(new Date()) => '"2013-08-08T17:13:10.751Z"'
JSON.stringify(NaN) => 'null'
JSON.stringify(Infinity) => 'null'
```

Definición y manipulación de objetos JSON

Además de serializar objetos JSON a partir de los que ya tengamos creados en JavaScript, podemos **definir** nuestros propios **objetos JSON** mediante la siguiente sintaxis:

- El **nombre** de la propiedad del objeto siempre va entre comillas "nombre": "valor"
- El **valor**:
  - Si es un número, booleano y `null`, van sin comillas.
  - Si es una cadena, va entre comillas dobles.
  - Si es otro objeto, va entre llaves.
  - Si es un array, va entre corchetes.

Vamos a ver varios ejemplos de objetos JSON: cómo se definen los objetos y cómo se accede a sus propiedades en JavaScript.

Ejemplo: objeto JSON sencillo, con propiedades que tienen como valores números y cadenas:

```
let objeto1 = {
    "nombre": "Ada",
    "nacimiento": 1815,
    "pais": "Reino Unido"
};
```

Para acceder a las propiedades, podemos utilizar dos notaciones:

```
objeto1.nombre;
objeto1["nombre"];
```

También se pueden recorrer todas las propiedades con un bucle `for ..in`:

```
for (let x in objeto1) {
    document.getElementById("demo").innerHTML +=
        x + ":" + objeto1[x] + "<br>";
}
```

Ejemplo: JSON con una propiedad que es un objeto (objeto anidado)

```
let objeto2 = {
    "nombre": "Ada",
    "nacimiento": 1815,
    "pais": "Reino Unido",
    "hijos": {
        "hijo1": "Anne Blunt",
        "hijo2": "Byron King-Noel",
        "hijo3": "Ralph King-Milbanke"
     }
}
```

Para acceder a las propiedades del objeto anidado, podemos utilizar dos opciones:

```
objeto2.hijos.hijo1;
objeto2.hijos["hijo1"];
```

Ejemplo: JSON con una propiedad que es un array

```
let objeto3 = {
    "nombre": "Ada",
    "nacimiento": 1815,
    "pais": "Reino Unido",
    "hijos": ["Anne Blunt", "Byron King-Noel",
              "Ralph King-Milbanke"]
}
```

Para acceder a los elementos del array, dos opciones:

```
objeto3.hijos[1];
objeto3["hijos"][1];
```

Enviar un objeto JSON al servidor PHP

En el cliente

Se puede enviar un objeto JSON al servidor desde el cliente. Para ello, habrá que convertirlo a cadena JSON utilizando el método `JSON.stringify(objetoJSON)`. Una vez convertido a cadena, se puede tratar como cualquier otro `string` utilizando GET o POST según los casos.

```
let puntos = document.getElementById("puntuacion").value;
let objeto = {
    "tabla": "alumnos",
    "valor": parseInt(puntos)
};
let parametros = JSON.stringify(objeto);
```

Para enviarlo con GET y `XMLHttpRequest`:

```
xhr.open("GET", "09_AJAX_JSON_BBDD.php?objeto=" + parametros);
xhr.send();
```

Si quisiéramos hacerlo con POST y `XMLHttpRequest`

```
xhr.open("POST", "09_AJAX_JSON_BBDD.php");
xhr.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
xhr.send("objeto=" + parametros);
```

Para enviarlo con GET y `Fetch`:

```
async function obtenerTexto(url) {
    let objeto = await fetch(url);
    …
}
obtenerTexto("09_AJAX_JSON_BBDD.php");
```

Si quisiéramos hacerlo con POST y `Fetch`:

```
let opciones = {
    method: "POST",
    body: "objeto=" + parametros,
    headers: {
        "Content-type": "application/x-www-form-urlencoded"
    }
};
async function obtenerTexto(url) {
    let objeto = await fetch(url, opciones);
    …
}
obtenerTexto("09_AJAX_JSON_BBDD.php");
```

En el servidor

El parámetro, se recibe como cadena. Utilizamos la función PHP `json_decode(cadena)`, para pasarla a objeto PHP. También depende de si nos han enviado los datos con GET o POST.

Si la hemos recibido con GET:

```php
<?php
$objeto = json_decode($_GET["objeto"],false);
?>
```

Si la hemos recibido con POST:

```php
<?php
$objeto = json_decode($_POST["objeto"],false);
?>
```

Recibir un objeto JSON del servidor PHP

En el servidor

En el servidor trabajamos con objetos PHP y antes de enviarlo al cliente, se pasa a cadena JSON utilizando la función `json_encode($objeto)`:

```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);
?>
```

En el cliente

Con `XMLHttpRequest` el objeto JSON se envía como una cadena de texto normal, por lo cual se recibe en la propiedad `responseText`. A partir de esa cadena, debemos obtener el objeto JSON utilizando el método `JSON.parse(cadena)`:

```javascript
let array = JSON.parse(this.responseText);
```

Con `Fetch`, el objeto JSON se extrae directamente con el método de `Response` `json()`:

```javascript
let array = await objeto.json();
```