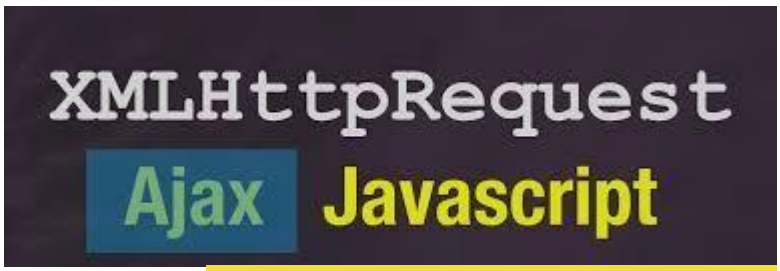
JavaScript



Comunicación asíncrona con el servidor

Contenido:

1. Breve historia de AJAX

La historia de AJAX está íntimamente relacionada con un objeto de programación llamado **`XMLHttpRequest`**. El origen de este objeto se remonta al año 2000, con productos como Exchange 2000, Internet Explorer 5 y Outlook WebAccess.

Todo comenzó en 1998, cuando **Alex Hopmann** y su equipo se encontraban desarrollando la entonces futura versión de Exchange 2000. El punto débil del servidor de correo electrónico era su cliente vía web, llamado OWA (*Outlook Web Access*).

Durante el desarrollo de OWA, se evaluaron dos opciones: un cliente formado sólo por páginas HTML estáticas que se recargaban constantemente y un cliente realizado completamente con HTML dinámico o DHTML. Alex Hopmann pudo ver las dos opciones y se decantó por la basada en DHTML. Sin embargo, para ser realmente útil a esta última le faltaba un componente esencial: "algo" que evitara tener que enviar continuamente los formularios con datos al servidor.

Motivado por las posibilidades futuras de OWA, Alex creó en un solo fin de semana la primera versión de lo que denominó XMLHTTP. La primera demostración de las posibilidades de la nueva tecnología fue un éxito, pero faltaba lo más difícil: incluir esa tecnología en el navegador Internet Explorer.

Si el navegador no incluía XMLHTTP de forma nativa, el éxito del OWA se habría reducido enormemente. El mayor problema es que faltaban pocas semanas para que se lanzara la última beta de Internet Explorer 5 previa a su lanzamiento final. Gracias a sus contactos en la empresa, Alex consiguió que su tecnología se incluyera en la librería MSXML que incluye Internet Explorer.

De hecho, el nombre del objeto (`XMLHTTP`) se eligió para tener una buena excusa que justificara su inclusión en la librería XML de Internet Explorer, ya que este objeto está mucho más relacionado con HTTP que con XML.

2. La primera aplicación

2.1. Código fuente

La aplicación AJAX completa más sencilla consiste en una adaptación del clásico *"Hola Mundo"*. En este caso, una aplicación JavaScript descarga un archivo del servidor y muestra su contenido sin necesidad de recargar la página.

Código fuente completo:

```
function descargaArchivo() {
    // Obtener la instancia del objeto XMLHttpRequest
    if(window.XMLHttpRequest) {
        peticion_http = new XMLHttpRequest();
    }
    else if(window.ActiveXObject) {// Para navegadores obsoletos
        peticion_http = new ActiveXObject("Microsoft.XMLHTTP");
    }

    // Preparar la funcion de respuesta
```

```
    peticion_http.onreadystatechange = muestraContenido;

    // Realizar peticion HTTP
    peticion_http.open('GET', 'http://localhost/holamundo.txt', true);
    peticion_http.send(null);

    }  window.onload = descargaArchivo;

    function muestraContenido() {
        if(peticion_http.readyState == 4) {
            if(peticion_http.status == 200) {
                alert(peticion_http.responseText);
            }
        }
    }
```

La aplicación AJAX del ejemplo anterior se compone de cuatro grandes bloques: instanciar el objeto **`XMLHttpRequest`**, preparar la función de respuesta, realizar la petición al servidor y ejecutar la función de respuesta.

Todas las aplicaciones realizadas con técnicas de AJAX deben instanciar en primer lugar el objeto **`XMLHttpRequest`**, que es el objeto clave que permite realizar comunicaciones con el servidor en segundo plano, sin necesidad de recargar las páginas.

La implementación del objeto `XMLHttpRequest` depende de cada navegador, por lo que es necesario emplear una discriminación sencilla en función del navegador en el que se está ejecutando el código:

```
if(window.XMLHttpRequest) { // Navegadores que siguen los estándares
    peticion_http = new XMLHttpRequest();
} else if(window.ActiveXObject) {  // Navegadoresobsoletos
    peticion_http = newActiveXObject("Microsoft.XMLHTTP");
}
```

Los navegadores actuales implementan el objeto `XMLHttpRequest` de forma nativa, por lo que se puede obtener a través del objeto `window`. Los navegadores obsoletos (Internet Explorer 6 y anteriores) implementan el objeto `XMLHttpRequest` como un objeto de tipo ActiveX.

Una vez obtenida la instancia del objeto `XMLHttpRequest`, se prepara la función que se encarga de procesar la respuesta del servidor. La propiedad `onreadystatechange` del objeto `XMLHttpRequest` permite indicar esta función directamente incluyendo su código mediante una función anónima o indicando una referencia a una función independiente. En el ejemplo anterior se indica directamente el nombre de la función:

```
peticion_http.onreadystatechange = muestraContenido;
```

El código anterior indica que cuando la aplicación reciba la respuesta del servidor, se debe ejecutar la función `muestraContenido()`. Como es habitual, la referencia a la función se indica mediante su nombre sin paréntesis, ya que de otro modo se estaría ejecutando la función y almacenando el valor devuelto en la propiedad `onreadystatechange`.

Después de preparar la aplicación para la respuesta del servidor, se realiza la petición HTTP al servidor:

```
peticion_http.open('GET', 'http://localhost/prueba.txt', true);
peticion_http.send(null);
```

Las instrucciones anteriores realizan el tipo de petición más sencillo que se puede enviar al servidor. En concreto, se trata de una petición de tipo `GET` simple que no envía ningún parámetro al servidor. La petición HTTP se crea mediante el método `open()`, en el que se incluye el tipo de petición (`GET`), la URL solicitada (`http://localhost/prueba.txt`) y un tercer parámetro que vale `true`.

Una vez creada la petición HTTP, se envía al servidor mediante el método `send()`. Este método incluye un parámetro que en el ejemplo anterior vale `null`. Más adelante se ven en detalle todos los métodos y propiedades que permiten hacer las peticiones al servidor. **Aquí solamente diremos que ese parámetro contendrá los datos que se enviarán al servidor en la petición. Antes era obligatorio ponerlo aunque no se enviaran datos. Ahora, si no se van a enviar datos ya no es necesario añadir *null* a la función *send()*.**

Por último, cuando se recibe la respuesta del servidor, la aplicación ejecuta de forma automática la función establecida anteriormente.

```
function muestraContenido() {
        if(peticion_http.readyState == 4) {
                if(peticion_http.status == 200) {
                alert(peticion_http.responseText); }
        }
}
```

La función `muestraContenido()` comprueba en primer lugar que se ha recibido la respuesta del servidor (mediante el valor de la propiedad `readyState`). Si se ha recibido alguna respuesta, se comprueba que sea válida y correcta (comprobando si el código de estado HTTP devuelto es igual a `200`). Una vez realizadas las comprobaciones, simplemente se muestra por pantalla el contenido de la respuesta del servidor (en este caso, el contenido del archivo solicitado) mediante la propiedad `responseText`.

2.2. Refactorizando la primera aplicación

A continuación, se muestra el código completo de la refactorización de la primera aplicación:

```
var READY_STATE_UNINITIALIZED=0;
var READY_STATE_LOADING=1;
var READY_STATE_LOADED=2;
var READY_STATE_INTERACTIVE=3;
var READY_STATE_COMPLETE=4;

var peticion_http;

function cargaContenido(url, metodo, funcion) {
    peticion_http = inicializa_xhr();
    if(peticion_http) {
        peticion_http.onreadystatechange = funcion;
```

```
        peticion_http.open(metodo, url, true);  peticion_http.send(null);
    }
}

function inicializa_xhr() {
    if(window.XMLHttpRequest) {
        return new XMLHttpRequest();
    } else if(window.ActiveXObject) {
        return new ActiveXObject("Microsoft.XMLHTTP"); }
}

function muestraContenido() {
    if(peticion_http.readyState == READY_STATE_COMPLETE) {
        if(peticion_http.status == 200) {
            alert(peticion_http.responseText);
        }
    }
}

function descargaArchivo() {
    cargaContenido("http://localhost/holamundo.txt", "GET",muestraContenido);
}

window.onload = descargaArchivo;
```

3. Métodos y propiedades del objeto XMLHttpRequest

El objeto `XMLHttpRequest` posee muchas otras propiedades y métodos diferentes a las manejadas por la primera aplicación de AJAX. A continuación, se incluye la lista completa de todas las propiedades y métodos del objeto y todos los valores numéricos de sus propiedades.

Las propiedades definidas para el objeto `XMLHttpRequest` son:

Propiedad	Descripción
`readyState`	**Valor numérico (entero) que almacena el estado de la petición.**
`responseText`	**El contenido de la respuesta del servidor en forma de cadena de texto. También se acepta ya la proriedad *"response"*.**
`responseXML`	**El contenido de la respuesta del servidor en formato XML. El objeto devuelto se puede procesar como un objeto DOM,**
`responseType`	**Valor de cadena enumerado que especifica el tipo de datos contenidos en la respuesta. También permite que el autor cambie el tipo de respuesta. Si se establece una cadena vacía al valor de responseType, se usa el valor predeterminado de "*text*". Por ejemplo, si ponemos:** **xhr = new XMLHttpRequest();** **xhr.responseType = "json";**

	, no tendremos que parsear (JSON.parse(xhr.response)) los datos, los tendremos directamente en formato json.
`status`	El código de estado HTTP devuelto por el servidor (`200` para una respuesta correcta, `404` para "No encontrado", `500` para un error de servidor, etc.).
`statusText`	El código de estado HTTP devuelto por el servidor en forma de cadena de texto: "OK", "Not Found", "Internal Server Error", etc.

Los valores definidos para la propiedad `readyState` son los siguientes:

Valor	Descripción
`0`	No inicializado (objeto creado, pero no se ha invocado el método open)
`1`	Cargando (objeto creado, pero no se ha invocado el método `send`)
`2`	Cargado (se ha invocado el método `send`, pero el servidor aún no ha respondido)
`3`	Interactivo (se han recibido algunos datos, aunque no se puede emplear la propiedad `responseText`)
`4`	Completo (se han recibido todos los datos de la respuesta del servidor)

Los métodos disponibles para el objeto `XMLHttpRequest` son los siguientes:

Método	Descripción
`abort()`	Detiene la petición actual
`getAllResponseHeaders()`	Devuelve una cadena de texto con todas las cabeceras de la respuesta del servidor.
`getResponseHeader("cabecera")`	Devuelve una cadena de texto con el contenido de la cabecera solicitada
`onreadystatechange`	Responsable de manejar los eventos que se producen. Se invoca cada vez que se produce un cambio en el estado de la petición HTTP. Normalmente es una referencia a una función JavaScript
`open("metodo", "url")`	Establece los parámetros de la petición que se realiza al servidor. Los parámetros necesarios son el método HTTP empleado y la URL destino (puede indicarse de forma absoluta o relativa)
`send(contenido)`	Realiza la petición HTTP al servidor
`setRequestHeader("cabecera", "valor")`	Permite establecer cabeceras personalizadas en la petición HTTP. Se debe invocar el método open()antes que setRequestHeader()

El método `open()` requiere dos parámetros (método HTTP y URL) y acepta de forma opcional otros tres parámetros. Definición formal del método `open()`:

```
open(string metodo, string URL [,boolean asincrono, string usuario, string password]);
```

Por defecto, las peticiones realizadas son asíncronas. Si se indica un valor `false` al tercer parámetro, la petición se realiza de forma síncrona, esto es, se detiene la ejecución de la aplicación hasta que se recibe de forma completa la respuesta del servidor.

No obstante, las peticiones síncronas son justamente contrarias a la filosofía de AJAX. El motivo es que una petición síncrona *congela* el navegador y no permite al usuario realizar ninguna acción hasta que no se haya recibido la respuesta completa del servidor. La sensación que provoca es que el navegador se ha *colgado* por lo que no se recomienda el uso de peticiones síncronas salvo que sea imprescindible.

Los últimos dos parámetros opcionales permiten indicar un nombre de usuario y una contraseña válidos para acceder al recurso solicitado.

Por otra parte, el método `send()`requiere de un parámetro que indica la información que se va a enviar al servidor junto con la petición HTTP. Se puede indicar como parámetro una cadena de texto, un array de bytes, un objeto XML o un objeto (JSON).

Por último, hay que saber que las únicas cabeceras que están permitidas establecer manualmente son:

- **`Accept`**: indica que tipo de contenido puede procesar el cliente. Por ejemplo: text/html, image/png, application/xml, */*, ...
- **`Accept-Language`**: indica qué lenguajes puede interpretar el cliente.
- **`Content-Language`**: se usa para describir los idiomas en los que está destinado que se entienda el documento, no quiere decir que el documento esté escrito en alguno de esos idiomas. Por ejemplo: es-ES, de-DE, en-EN.
- **`Content-Type`**: **le dice al cliente cuál es realmente el tipo del contenido devuelto. Es la más usual tener que establecer en la petición.**

Los únicos valores permitidos de la cabecera Content-Type son:

- **`application/x-www-form-urlencoded`**: Los valores son codificados en tuplas llave-valor separadas por '&', con un '=' entre la llave y el valor. Caracteres no-Alfanuméricos en ambas (llaves, valores) son codificadas con '%'. Esta es la razón por la cual este tipo no es adecuado para usarse con datos binarios, como imágenes por ejemplo, (usar **multipart/form-data** en su lugar).
- **`multipart/form-data`**: Se puede utilizar para enviar los valores de un formulario HTML completo del navegador al servidor.
- **`text/plain`**: Para recibir texto plano. También se puede usar text/html para obtener texto o código html.
- **`application/json`**: Para enviar código json.

Ejercicio:

A partir del ejemplo anterior, añadir el código JavaScript necesario para que:

1. Al cargar la página, el cuadro de texto debe mostrar por defecto la URL de la propia página.
2. Al pulsar un botón "Mostrar Contenidos", se debe descargar mediante peticiones AJAX el contenido correspondiente a la URL introducida por el usuario.
3. Se debe mostrar en todo momento el estado en el que se encuentra la petición (No inicializada, cargando, completada, etc.)
4. Mostrar el contenido de todas las cabeceras de la respuesta del servidor en la zona "Cabeceras HTTP de la respuesta del servidor".

4. Interacción con el servidor

4.1. Envío de parámetros con la petición HTTP

Hasta ahora, el objeto `XMLHttpRequest` se ha empleado para realizar peticiones HTTP sencillas. Sin embargo, las posibilidades que ofrece el objeto `XMLHttpRequest` son muy superiores, ya que también permite el envío de parámetros junto con la petición HTTP.

El objeto `XMLHttpRequest` puede enviar parámetros tanto con el método **GET** como con el método **POST** de HTTP. **En ambos casos, los parámetros se envían como una serie de pares clave/valor concatenados por símbolos `&`.** El siguiente ejemplo muestra una URL que envía parámetros al servidor mediante el método GET:

`http://localhost/aplicacion?parametro1=valor1&parametro2=valor2&parametro3=valor3`

La principal diferencia entre ambos métodos es que mediante el método POST los parámetros se envían en el cuerpo de la petición y mediante el método GET los parámetros se concatenan a la URL accedida. El método GET se utiliza cuando se accede a un recurso que depende de la información proporcionada por el usuario. El método POST se utiliza en operaciones que crean, borran o actualizan información.

Técnicamente, el método GET tiene un límite en la cantidad de datos que se pueden enviar. Si se intentan enviar más de 512 bytes mediante el método GET, el servidor devuelve un error con código `414` y mensaje `Request-URI Too Long`(*"La URI de la petición es demasiado larga"*).

Cuando se utiliza un elemento `<form>` de HTML, al pulsar sobre el botón de envío del formulario, se crea automáticamente la cadena de texto que contiene todos los parámetros que se envían al servidor. Sin embargo, el objeto `XMLHttpRequest` no dispone de esa posibilidad y la cadena que contiene los parámetros se debe construir manualmente.

A continuación, se incluye un ejemplo del funcionamiento del envío de parámetros al servidor. Se trata de un formulario con tres campos de texto que se validan en el servidor mediante AJAX. El código HTML también incluye un elemento `<div>` vacío que se utiliza para mostrar la respuesta del servidor:

```
<form>
```

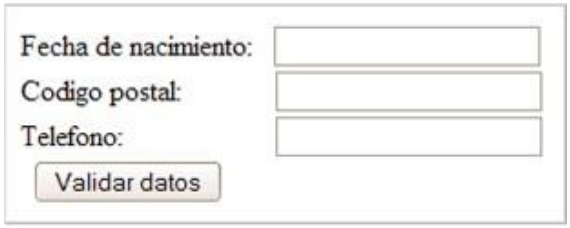
```
    <label for="fecha_nacimiento">Fecha de nacimiento:</label>
    <input type="text" id="fecha_nacimiento" name="fecha_nacimiento" /><br/>
    <label for="codigo_postal">Codigo postal:</label>
    <input type="text" id="codigo_postal" name="codigo_postal" /><br/>

    <label for="telefono">Telefono:</label>
    <input type="text" id="telefono" name="telefono" /><br/>

    <input type="button" value="Validar datos" />
</form>

<div id="respuesta"></div>
```

El código anterior produce la siguiente página:



El código JavaScript necesario para realizar la validación de los datos en el servidor se muestra a continuación:

```
var READY_STATE_COMPLETE=4;
var peticion_http = null;
 function inicializa_xhr() {
    if(window.XMLHttpRequest) {
        return new XMLHttpRequest();
    }
    else if(window.ActiveXObject) {
            return new ActiveXObject("Microsoft.XMLHTTP");
    }
}

function crea_query_string() {
    var fecha = document.getElementById("fecha_nacimiento");
    var cp = document.getElementById("codigo_postal");
    var telefono = document.getElementById("telefono");
    return "fecha_nacimiento=" + encodeURIComponent(fecha.value) + "&codigo_postal="
    + encodeURIComponent(cp.value) + "&telefono=" + encodeURIComponent(telefono.value) + "&nocache="
+ Math.random();
}

function valida() {
    peticion_http = inicializa_xhr();
    if(peticion_http) {
```

```
        peticion_http.onreadystatechange = procesaRespuesta;
        peticion_http.open("POST", "http://localhost/validaDatos.php");

        peticion_http.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
        var query_string = crea_query_string();
        peticion_http.send(query_string);
    }
}

function procesaRespuesta() {
    if(peticion_http.readyState == READY_STATE_COMPLETE) {
        if(peticion_http.status == 200) {
            document.getElementById("respuesta").innerHTML = peticion_http.responseText;
        }
    }
}
```

La clave del ejemplo anterior se encuentra en estas dos líneas de código:

```
peticion_http.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");

peticion_http.send(query_string);
```

En primer lugar, si no se establece la cabecera `Content-Type` correcta, el servidor descarta todos los datos enviados mediante el método POST. De esta forma, al programa que se ejecuta en el servidor no le llega ningún parámetro. Así, **para enviar parámetros mediante el método POST, es obligatorio incluir la cabecera `Content-Type`** mediante la siguiente instrucción:

```
peticion_http.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
```

Por otra parte, el método `send()` es el que se encarga de enviar los parámetros al servidor. En todos los ejemplos anteriores se utilizaba la instrucción `send(null)` para indicar que no se envían parámetros al servidor. Sin embargo, en este caso la petición sí que va a enviar los parámetros.

Tenemos una función que crea una cadena con todos los campos del formulario y los concatena junto con el nombre de cada parámetro para formar la cadena de texto que se envía al servidor. El uso de la función **`encodeURIComponent()`** es imprescindible para evitar problemas con algunos caracteres especiales.

La función `encodeURIComponent()` reemplaza todos los caracteres que no se pueden utilizar de forma directa en las URL por su representación hexadecimal. Las letras, números y los caracteres `- _ . ! ~ * ' ( )` no se modifican, pero todos los demás caracteres se sustituyen por su equivalente hexadecimal (espacios en blanco, acentos, ...).

JavaScript incluye una función contraria llamada **`decodeURIComponent()`** y que realiza la transformación inversa. Además, también existen las funciones `encodeURI()` y `decodeURI()`que codifican/decodifican una URL completa. La principal diferencia entre **`encodeURIComponent()` y `encodeURI()`**es que esta última no codifica los caracteres `; / ? : @ & = + $ , #`:

Por último, la función `crea_query_string()` añade al final de la cadena un parámetro llamado **`nocache`** y que contiene un número aleatorio (creado mediante el método **`Math.random()`**). Añadir un parámetro aleatorio adicional a las peticiones GET y POST es una de las estrategias más utilizadas para evitar problemas con la caché de los navegadores. Como cada petición varía al menos en el valor de uno de los parámetros, el navegador está obligado siempre a realizar la petición directamente al servidor y no utilizar su cache.

En las aplicaciones reales, las **validaciones de datos** mediante AJAX sólo se utilizan en el caso de validaciones complejas que no se pueden realizar mediante el uso de código JavaScript básico. En general, las validaciones complejas requieren el uso de bases de datos: comprobar que un nombre de usuario no esté previamente registrado, comprobar que la localidad se corresponde con el código postal indicado, etc.

Ejercicio:

Un ejemplo de validación compleja es la que consiste en comprobar si un nombre de usuario escogido está libre o ya lo utiliza otro usuario. Como es una validación que requiere el uso de una base de datos muy grande, no se puede realizar en el navegador del cliente. Utilizando las técnicas mostradas anteriormente:

1. Crear un script que compruebe con AJAX y la ayuda del servidor si el nombre escogido por el usuario está libre o no.
2. El script del servidor se llama compruebaDisponibilidad.php y el parámetro que contiene el nombre se llama login. Se usará el array PHP $_GET[] o $_POST[] para recibir los parámetros en el servidor.
3. La respuesta del servidor es 1 o 0, en función de si el nombre de usuario está libre y se puede utilizar o ya ha sido ocupado por otro usuario.
4. A partir de la respuesta del servidor, mostrar un mensaje al usuario indicando el resultado de la comprobación.

5. Envío de parámetros mediante XML

La flexibilidad del objeto `XMLHttpRequest` permite el envío de los parámetros por otros medios alternativos a la tradicional *query string*. De esta forma, si la aplicación del servidor así lo requiere, es posible realizar una petición al servidor enviando los parámetros en formato XML.

A continuación, se modifica el ejemplo anterior para enviar los datos del usuario en forma de documento XML. En primer lugar, se modifica la llamada a la función que construye la *query string*:

```
function valida() {
peticion_http = inicializa_xhr();

if(peticion_http) {
  peticion_http.onreadystatechange = procesaRespuesta;
  peticion_http.open("POST", "http://localhost/validaDatos.php", true);
  var parametros_xml = crea_xml();
  peticion_http.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
```

```
  peticion_http.send(parametros_xml);
}
}
```

Seguidamente, se crea la función `crea_xml()`que se encarga de construir el documento XML que contiene los parámetros enviados al servidor:

```
function crea_xml() {
  var fecha = document.getElementById("fecha_nacimiento");
  var cp = document.getElementById("codigo_postal");
  var telefono = document.getElementById("telefono");

   var xml = "<parametros>";     xml = xml + "<fecha_nacimiento>" +
  fecha.value + "<\/fecha_nacimiento>";       xml = xml + "<codigo_postal>" +
  cp.value + "<\/codigo_postal>";     xml = xml + "<telefono>" +
  telefono.value + "<\/telefono>"; xml = xml +
  "<\/parametros>";

  return xml;
}
```

El código de la función anterior emplea el carácter "\" en el cierre de todas las etiquetas XML. El motivo es que las etiquetas de cierre XML y HTML (al contrario que las etiquetas de apertura) se interpretan en el mismo lugar en el que se encuentran, por lo que si no se incluyen esos caracteres \ el código no validaría siguiendo el estándar HTML de forma estricta.

El método `send()` del objeto `XMLHttpRequest` permite el envío de una cadena de texto y de un documento XML. Sin embargo, en el ejemplo anterior se ha optado por una solución intermedia: una cadena de texto que representa un documento XML. El motivo es que no existe a día de hoy un método robusto y que se pueda emplear en la mayoría de navegadores para la creación de documentos XML completos.

Para procesar estos datos más fácilmente en PHP, convertiremos el string XML a un array asociativo de la siguiente forma:

```
$xmlstr = file_get_contents('php://input');
$data = new SimpleXMLElement($xmlstr);

$telefono = $data->parametros[0]->telefono;
```

5.1. Procesando respuestas XML

Además del envío de parámetros en formato XML, el objeto `XMLHttpRequest` también permite la recepción de respuestas de servidor en formato XML. Una vez obtenida la respuesta del servidor mediante la propiedad **`petición_http.responseXML`**, es posible procesarla empleando los métodos DOM de manejo de documentos XML/HTML.

En este caso, se modifica la respuesta del servidor para que no sea un texto sencillo, sino que la respuesta esté definida mediante un documento XML:

```
<respuesta>
        <mensaje>...</mensaje>
        <parametros>
                <telefono>...</telefono>
                <codigo_postal>...</codigo_postal>
                <fecha_nacimiento>...</fecha_nacimiento>
        </parametros>
</respuesta>
```

La respuesta del servidor incluye un mensaje sobre el éxito o fracaso de la operación de validación de los parámetros y además incluye la lista completa de parámetros enviados al servidor.

La función encargada de procesar la respuesta del servidor se debe modificar por completo para tratar el nuevo tipo de respuesta recibida:

```
function procesaRespuesta() {

if(peticion_http.readyState == READY_STATE_COMPLETE){
    if(peticion_http.status == 200) {
       var documento_xml = peticion_http.responseXML;
       var root = documento_xml.getElementsByTagName("respuesta")[0];
       var mensajes = root.getElementsByTagName("mensaje")[0];
       var mensaje = mensajes.firstChild.nodeValue;
       var parametros = root.getElementsByTagName("parametros")[0];
       var tlf =parametros.getElementsByTagName("telefono")[0].firstChild.nodeValue;


       var f_nac = parametros.getElementsByTagName("fecha_nacimiento")[0].innerHTML;
       var cp = parametros.getElementsByTagName("codigo_postal")[0].firstChild.nodeValue;
       document.getElementById("respuesta").innerHTML = mensaje + "<br/>" + "Fecha nacimiento = " +
f_nac + "<br/>" + "Codigo postal = " + cp+ "<br/>" + "Telefono = " + tlf;
        }
    }
}
```

El primer cambio importante es el de obtener el contenido de la respuesta del servidor. Hasta ahora, siempre se utilizaba la propiedad `responseText`, que devuelve el texto simple que incluye la respuesta del servidor. Cuando se procesan respuestas en formato XML, se debe utilizar la propiedad **`responseXML`**.

El valor devuelto por `responseXML` es un documento XML que contiene la respuesta del servidor. Como se trata de un documento XML, ***es posible utilizar con sus contenidos todas las funciones DOM que se vieron en el capítulo correspondiente a DOM***.

Aunque el manejo de repuestas XML es mucho más pesado y requiere el uso de numerosas funciones DOM, su utilización se hace imprescindible para procesar respuestas muy complejas o respuestas recibidas por otros sistemas que exportan sus respuestas internas a un formato estándar XML.

El mecanismo para obtener los datos varía mucho según cada documento XML, pero en general, se trata de obtener el valor almacenado en algunos elementos XML que a su vez pueden ser descendientes de otros elementos.

Ejercicio:

Normalmente, cuando se valida la disponibilidad de un nombre de usuario, se muestra una lista de valores alternativos en el caso de que el nombre elegido no esté disponible. Modificar el ejercicio de comprobación de disponibilidad de los nombres para que permita mostrar una serie de valores alternativos devueltos por el servidor.

El script del servidor se llama `compruebaDisponibilidadXML.php` y el parámetro que contiene el nombre se llama `login`. La respuesta del servidor es un documento XML con la siguiente estructura:

Si el nombre de usuario está libre:

```
<respuesta>
	<disponible>si</disponible>
</respuesta>
```

Si el nombre de usuario está ocupado:

```
<respuesta>
	<disponible>no</disponible>
	<alternativas>
		<login>...</login>
		<login>...</login>
		...
		<login>...</login>
	</alternativas>
</respuesta>
```

Los nombres de usuario alternativos se deben mostrar en forma de lista de elementos (`<ul></ul>`).

Modificar la lista anterior para que muestre enlaces para cada uno de los nombres alternativos. Al pinchar sobre el enlace de un nombre alternativo, se copia en el cuadro de texto del login del usuario.

6. Recepción y envío con JSON

Aunque el formato XML está soportado por casi todos los lenguajes de programación, por muchas aplicaciones y es una tecnología madura y probada, en algunas ocasiones es más útil intercambiar información con el servidor en formato JSON.

JSON es un formato mucho **más compacto y ligero que XML**. Además, es mucho **más fácil de procesar en el navegador** del usuario. Afortunadamente, cada vez existen más utilidades para procesar y generar el formato JSON en los diferentes lenguajes de programación del servidor (PHP, Java, C#, etc.)

El ejemplo mostrado anteriormente para procesar las respuestas XML del servidor se puede reescribir utilizando respuestas JSON. En este caso, la respuesta que genera el servidor es mucho más concisa:

```
{ mensaje: "...", parametros: {telefono: "...", codigo_postal: "...",
fecha_nacimiento: "..." }
}
```

Considerando el nuevo formato de la respuesta, es necesario modificar la función que se encarga de procesar la respuesta del servidor:

```
function procesaRespuesta() {
    if(http_request.readyState == READY_STATE_COMPLETE) {
        if(http_request.status == 200){
            var respuesta_json = http_request.responseText;
            var objeto_json = JSON.parse(respuesta_json);
            var mensaje = objeto_json.mensaje;

            var telefono = objeto_json.parametros.telefono;
            var fecha_nacimiento = bjeto_json.parametros.fecha_nacimiento;
            var codigo_postal = objeto_json.parametros.codigo_postal;
                document.getElementById("respuesta").innerHTML = mensaje + "<br>" +
            "Fecha nacimiento = " + fecha_nacimiento + "<br>" + "Codigo postal = "
            + codigo_postal+ "<br>" + "Teléfono = " + telefono;
        }
    }
}
```

La respuesta JSON del servidor se obtiene mediante la propiedad `responseText`.

Sin embargo, esta propiedad solamente devuelve la respuesta del servidor en forma de cadena de texto. Para trabajar con el código JSON devuelto, **se debe transformar esa cadena de texto en un objeto JSON**. La forma más sencilla de realizar esa conversión es mediante la función JSON.**`parse()`.**

Una vez realizada la transformación, el objeto JSON ya permite acceder a sus métodos y propiedades mediante la notación de puntos tradicional. Comparado con las respuestas XML, este procedimiento permite acceder a la información devuelta por el servidor de forma mucho más simple:

```
// Con JSON
var f_nac = objeto_json.parametros.fecha_nacimiento;

// Con XML var parametros =
root.getElementsByTagName("parametros")[0];
var f_nac=parametros.getElementsByTagName("fecha_nacimiento")[0].firstChild.nodeValue;
```

También es posible el envío de los parámetros en formato JSON. Para ello tendremos que transformar el objeto JSON en una cadena de texto que será la que se envía al servidor. Se emplea el método **`stringify()`** para realizar la transformación:

```
var objeto_json = JSON.stringify(objeto);

xhr.send(objeto_json);
```

La mayoría de los lenguajes de programación incluyen ya librerías para codificar y decodificar JSON, como por ejemplo la librería *org.json.JSONObject* de Java.

Ejercicio:

Rehacer el ejercicio anterior para procesar respuestas del servidor en formato JSON. Los cambios producidos son:

1) El script del servidor se llama `compruebaDisponibilidadJSON.php` y el parámetro que contiene el nombre se llama `login`.

2) La respuesta del servidor es un objeto JSON con la siguiente estructura: El nombre de usuario está libre:

```
{ disponible: "si" }
```

El nombre de usuario está ocupado:

```
{ disponible: "no", alternativas: ["...", "...", ..., "..."] }
```

Para enviar datos en JSON, se podría hacer de la siguiente forma:

```
var myjson = { key: "valor", key1: "valor1", key2: "valor2" }; var
objetoXHR = new XMLHttpRequest();

// Establecemos la cabecera Content-Type apropiada
ajax_request.setRequestHeader("Content-Type", "application/json; charset=UTF-8");
// Enviar la solicitud
ajax_request.send( JSON.stringify(myjson) );
```

Pero PHP seguirá esperando que el cuerpo de la solicitud sea un query string. Al recibir un cuerpo con otro formato, no podremos acceder desde PHP a esos datos en la super global $_POST como es habitual. Tendremos que acceder directamente al cuerpo de la solicitud y decodificar el JSON:

```php
// Obtenemos el json enviado
$data = file_get_contents('php://input');

// Los convertimos en un array
$data = json_decode( $data, true );
```

Ejercicio:

Realiza una página web de selección origen/destino para la compra de billetes de avión.

Orígenes y destinos posibles:

- Madrid: Barcelona, Valencia, Sevilla.
- Barcelona: Madrid, Granada.
- Valencia: Madrid.

La página web estará conformada por:

1. Una lista desplegable donde se encuentre el listado de orígenes posibles.
2. Una vez seleccionado uno de los orígenes, se llamará a la función JavaScript con la ciudad de origen seleccionada.
3. Desde el servidor se retornará (de manera estructurada) el listado de ciudades destino.
4. Desde JavaScript se cargarán las ciudades en la lista desplegable destino.

XMLHttpRequest síncrono está en proceso de ser eliminado del estándar web, pero este proceso puede llevar varios años.

7. Seguridad

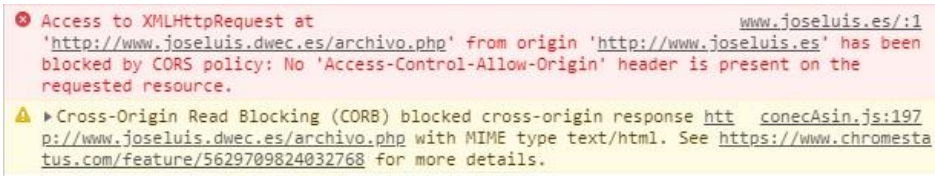
La ejecución de aplicaciones JavaScript puede suponer un riesgo para el usuario que permite su ejecución. Por este motivo, los navegadores restringen la ejecución de todo código JavaScript a un entorno de ejecución limitado y prácticamente sin recursos ni permisos para realizar tareas básicas.

Las aplicaciones JavaScript no pueden leer ni escribir ningún archivo del sistema en el que se ejecutan. Tampoco pueden establecer conexiones de red con dominios distintos al dominio en el que se aloja la aplicación JavaScript. Además, un script sólo puede cerrar aquellas ventanas de navegador que ha abierto ese script.

La restricción del acceso a diferentes dominios es más restrictiva de lo que en principio puede parecer. El problema es que los navegadores emplean un método demasiado simple para diferenciar entre dos dominios ya que no permiten ni subdominios ni otros protocolos ni otros puertos.

Por ejemplo, si el código JavaScript se descarga desde la siguiente URL: http://www.ejemplo.com/scripts/codigo.js, las funciones y métodos incluidos en ese código no pueden acceder a los recursos contenidos en los siguientes archivos:

- http://www.ejemplo.com:8080/scripts/codigo2.js
- https://www.ejemplo.com/scripts/codigo2.js
- http://192.168.0.1/scripts/codigo2.js
- http://scripts.ejemplo.com/codigo2.js



7.1. CORS

El W3C (Grupo de trabajo de Aplicaciones Web), recomienda el nuevo mecanismo **de Intercambio de Recursos de Origen Cruzado** (CORS, por sus siglas en inglés). CORS da controles de acceso a dominios cruzados para servidores web y transferencia segura de datos en dominios cruzados entre navegadores y servidores Web. Los exploradores modernos utilizan CORS en un contenedor API (como XMLHttpRequest o Fetch) para ayudar a mitigar los riesgos de solicitudes HTTP de origen cruzado.

CORS es el acrónimo de Cross-origin resource sharing. Si trabajamos con CORS, que es lo más actual, requerirá una pequeña configuración del servidor. Además, debemos tener presente, que los navegadores antiguos no soportan CORS.

Una solicitud HTTP de origen cruzado es una que se hace para:

- Un dominio diferente (por ejemplo, de example.com a amazondomains.com)
- Un subdominio diferente (por ejemplo, de example.com a petstore.example.com)
- Un puerto diferente (por ejemplo, de example.com a example.com:10777)
- Un protocolo diferente (por ejemplo, de https://example.com a http://example.com)

Si tu aplicación está en www.example.com y quieres obtener datos de www.example2.com, el host example2 permitirá peticiones de *example* añadiendo una cabecera:

```
Access-Control-Allow-Origin:http://www.example.com
```

Para permitir hacer peticiones a cualquier dominio podemos usar esta otra cabecera:

```
Access-Control-Allow-Origin: *
```

`Access-Control-Allow-Origin` se puede añadir a un solo recurso o a todo el dominio.

El script que utilizaríamos en el cliente podría ser similar al siguiente:

```
peticion_http.setRequestHeader("Access-Control-Allow-Origin", "*");
```

En PHP estableceríamos la cabecera así:

```
<?php  header('Access-Control-Allow-Origin:
*');  ?>
```

Existen una serie de cabeceras, además de Access-Control-Allow-Origin, que podemos configurar con CORS. Las más básicas son:

```
Access-Control-Allow-Origin: https://foo.app.moxio.com
Access-Control-Allow-Credentials: true
Access-Control-Allow-Methods: POST
Access-Control-Allow-Headers: Content-Type
```

En el ejemplo anterior, se establecen las cabeceras CORS de forma que el origen https://foo.app.moxio.com puede realizar una solicitud POST, se pueden incluir cookies y podemos enviar el encabezado Content-Type.

Podríamos hacer un envío desde JavaScript para que permita esta política de la siguiente forma:

```
var http_request;

    http_request = new XMLHTTPRequest();
    http_request.onreadystatechange = function () { /* .. */ };
    http_request.open("POST", "https://.........");

    http_request.withCredentials = true;
    http_request.setRequestHeader("Content-Type", "application/json");
    http_request.send({ 'request': "authentication token" });
```

Estamos enviando una solicitud POST que contiene JSON e incluiremos nuestras cookies, credenciales, etc. Produce una solicitud con estos encabezados:

```
Origin: https://......
Access-Control-Request-Method: POST
Access-Control-Request-Headers: Content-Type
```

El servidor puede incluir algunos encabezados Access-Control- * dentro de la respuesta para indicar si la solicitud que sigue se permitirá o no. Éstas son:

- Access-Control-Allow-Origin: el origen que tiene permiso para realizar la solicitud, o * si se puede realizar una solicitud desde cualquier origen.
- Access-Control-Allow-Methods: indica qué métodos HTTP están permitidos para solicitudes de origen cruzado. Si permite todos los métodos HTTP, entonces está bien establecer el valor en algo como Access-Control-Allow-Methods: GET, PUT, POST, DELETE, HEAD.
- Access-Control-Allow-Headers: una lista separada por comas de los encabezados personalizados que se pueden enviar.

- Access-Control-Max-Age: la duración máxima que la respuesta en caché antes de realizar otra llamada.

En el servidor podemos recoger la petición de la siguiente forma:

```
if (isset($_SERVER["HTTP_ORIGIN"]) === true) {
    $origin = $_SERVER["HTTP_ORIGIN"];
    $allowed_origins = array(
              "http://public.app.moxio.com",
              "https://foo.app.moxio.com",
              "https://lorem.app.moxio.com"
    );
    if (in_array($origin, $allowed_origins, true) === true) {
      header('Access-Control-Allow-Origin: ' . $origin);
      header('Access-Control-Allow-Credentials: true');
      header('Access-Control-Allow-Methods: POST');
      header('Access-Control-Allow-Headers: Content-Type');
    }
    if ($_SERVER["REQUEST_METHOD"] === "POST") {
              …
    }
}
```

Hay varias cosas importantes que muestra este ejemplo:

1. http y https son orígenes diferentes.
2. No permitimos todos los orígenes. Puede establecer Allow-Origin en '*' para permitir todos los orígenes. Esto puede ser útil si tiene una API pública.
3. No exponemos la lista de orígenes permitidos. Puede configurar Allow-Origin en una lista de dominios separados por comas, pero esta es más información de la que necesita la solicitud. Dado que CORS es principalmente una característica de seguridad, tiene sentido establecerlo lo más restrictivo posible.
4. Para la solicitud de verificación previa solo necesitamos devolver la política CORS, no es necesario procesar la solicitud por completo.

8. Uso de la API fetch.

El API fetch es un nuevo estándar que viene a dar una alternativa para interactuar por HTTP, con un diseño moderno, basado en promesas, con mayor flexibilidad y capacidad de control a la hora de realizar llamadas al servidor. También está disponible en **Node**, por lo que podemos utilizarlo de forma isomórfica, es decir, tanto en cliente como en servidor.

8.1. Funcionamiento básico

Una de las características más importantes del API **fetch es que utiliza promesas**, es decir, devuelve un objeto con dos métodos, uno then() y otro catch() a la que pasaremos una función que será invocada cuando se obtenga la respuesta o se produzca un error.

Aquí hay que aclarar un punto con los errores: si se devuelve un código HTTP correspondiente a un error no se ejecutará el catch(), ya que se ha obtenido una respuesta válida, por lo que se ejecutará el then(). Sólo si hay un error de red o de otro tipo se ejecutará el catch().

Otro aspecto importante que hay que comprender es que para obtener el body o cuerpo del mensaje devuelto por el servidor **deberemos obtener una segunda promesa por medio de los métodos del objeto *Response***. Por ello, será muy habitual ver dos promesas encadenadas, una para el fetch() y otra con el retorno del método que utilicemos para obtener el body.

En los ejemplos vamos a utilizar las opciones que nos ofrece https://httpbin.org/ para comprobar el funcionamiento de nuestros clientes HTTP.

Vamos a verlo paso a paso:

```
// Uso de API Fetch básico:
fetch('https://httpbin.org/ip')
.then(function(response) {
  return response.text();
})
.then(function(data) {
    console.log('datos = ', data);
    data.text()
    .then (function(data) {
      //...
    })
    .catch(function(err) {
      console.error(err);
    })
})
.catch(function(err) {
    console.error(err);
});
```

¿Qué hemos hecho?:

- **hemos llamado a `fetch()`** con la URL a la que queremos acceder como parámetro
- esta llamada nos devuelve una **promesa**
- el método `then()` de esa promesa **nos entrega un objeto `response`**
- del objeto `response` **llamamos al método `text()`** para obtener el cuerpo retornado en forma de texto
- nos devuelve **otra promesa** que se resolverá cuando se haya obtenido el contenido
- el método `then()` de esa promesa **recibe el cuerpo** devuelto por el servidor en formato de texto • hemos incluido un `catch()` por si se produce algún error

Hay que destacar que, la respuesta del servidor en la primera promesa puede ser de varios tipos, en concreto:

- **arrayBuffer() - devuelve la respuesta como un objeto *ArrayBuffer* (representación binaria de datos de bajo nivel),**
- **blob() - devuelve la respuesta como Blob (datos binarios tipados).**
- **json() - convierte la respuesta como un JSON.**
- **text() - lee y devuelve la respuesta en formato texto.**
- **formData() - Devuelve la respuesta como un objeto *FormData*. Aquí puedes ver una guía para su uso. Más info.**

La llamada a cualquiera de estos métodos devuelve una nueva promesa. Así que, para recuperar los datos tendremos que concatenar la respuesta con un ***".then()"***. De esta forma, el ejemplo anterior quedaría de la siguiente manera:

Aquí tienes algunos ejemplos:

1.- JSON: Por ejemplo, si obtenemos un objeto de tipo JSON con los últimos commits de GitHub:

```
let url = 'https://api.github.com/repos/javascript-
tutorial/en.javascript.info/commits';
let response = await fetch(url);
let commits = await response.json(); // leer respuesta del cuerpo y devolver como JSON
alert(commits[0].author.login);

//.. o de forma equivalente:

fetch('https://api.github.com/repos/javascript-tutorial/en.javascript.info/commits')
  .then(response => response.json())
  .then(commits => alert(commits[0].author.login));
```

2.- Como demostración de una lectura en formato binario, hagamos un fetch y mostremos una imagen del logotipo de "especificación fetch" (ver capítulo Blob para más detalles acerca de las operaciones con Blob):

```
let response = await fetch('/article/fetch/logo-fetch.svg');

let blob = await response.blob(); // se guarda en un objeto tipo Blob

// crear <img> para el logo
let img = document.createElement('img');
img.style = 'position:fixed;top:10px;left:10px;width:100px';
document.body.append(img);

// mostrar
img.src = URL.createObjectURL(blob);

setTimeout(() => { // ocultar luego de tres segundos
  img.remove();
```

```
  URL.revokeObjectURL(img.src);
}, 3000);
```

8.2. Opciones de fetch()

Quizás no nos hayamos dado cuenta, pero en el ejemplo anterior no hemos indicado que método teníamos que utilizar, simplemente hemos pasado la URL y se considera que queremos utilizar el método GET que es el valor por defecto. La forma de configurar esta llamada es utilizar el segundo parámetro de fetch(), donde pasaremos un objeto con las opciones.

En este nuevo ejemplo vamos a llamar a https://httpbin.org/post, un servicio que nos va a devolver un JSON con lo que enviemos con un método POST y alguna información adicional.

```
//Segundo parámetro de fetch():
```

```
fetch('https://httpbin.org/post', {
    method: 'POST',
    headers: {'Content-Type': 'application/x-www-form-urlencoded'},
    body: 'a=1&b=2'
})
.then(function(response) {
    console.log('response =', response);
    return response.json();
})
.then(function(data) {
    console.log('data = ', data);
})
.catch(function(err) {
    console.error(err);
});
```

Como podemos ver, hemos configurado el método, una cabecera y el cuerpo de la llamada.

Vamos a ver una variante donde se envían datos en formato JSON y se indica que no se utilice la caché:

```
// Enviando JSON:

fetch('https://httpbin.org/post',{
    method: 'POST',
    headers: {'Content-Type': 'application/json'},
    body: JSON.stringify({"a": 1, "b": 2}),
    cache: 'no-cache'
})
.then(function(response) {
    return response.json();
})
```

```
.then(function(data) {
    console.log('data = ', data);     })
.catch(function(err) {
    console.error(err);
});
```

Las opciones que podemos configurar son:

- **method**: método a utilizar.
- **headers**: cabeceras que se deben enviar (ver objeto Headers).
- **body**: cuerpo que se envía al servidor, que puede ser una cadena, un objecto Blob, BufferSource, FormData o URLSearchParams.
- **mode**: modo de la solicitud: **'cors', 'no-cors', 'same-origin', 'navigate'**.

  1. cors (por defecto): Permite solicitudes a otros orígenes (dominios diferentes) **siempre que el servidor tenga configurado CORS correctamente**. Es útil para consumir APIs externas. Si el servidor no permite CORS, el navegador bloqueará la solicitud.

  2. no-cors: Solo permite realizar solicitudes a otros orígenes **sin acceder a la respuesta**. Se usa para enviar datos a servidores externos sin importar la respuesta (por ejemplo, con POST).

  3. same-origin: Solo permite solicitudes **al mismo origen** (mismo dominio, protocolo y puerto). Si intentas acceder a otro origen, la solicitud será bloqueada.

  4. navigate: Se usa para peticiones de navegación de documentos (cuando se carga una página o se sigue un enlace). No se puede usar en fetch, ya que está reservado para el navegador.
- **credentials**:

  1. "omit" (Por defecto en solicitudes cross-origin): No envía credenciales (cookies, encabezados Authorization, etc.). **No acepta credenciales en la respuesta**. Útil cuando no necesitas autenticación y quieres evitar enviar cookies accidentalmente.

  2. "same-origin" (Por defecto en solicitudes same-origin): Solo envía credenciales si el origen es el mismo (mismo dominio, protocolo y puerto). Si el destino es otro origen, se comporta como "omit" (no envía credenciales).

  3. "include": Siempre envía credenciales, incluso en solicitudes **cross-origin**. Para que funcione en CORS, el servidor debe tener la cabecera: Access-Control-Allow-Credentials: true
- **cache**:

  1. "default": Usa la estrategia de caché predeterminada del navegador. Generalmente, intenta usar la caché primero si es posible. Equivalente a "force-cache" en navegadores que soportan caché agresiva.

2. "no-store": No usa ni almacena en caché. Siempre obtiene una nueva respuesta desde el servidor. Ideal para datos altamente dinámicos como precios o estados en tiempo real.

3. "reload": Omite la caché y va directamente al servidor. **No almacena la respuesta en caché**. Similar a "no-store", pero si otra pestaña ya había almacenado la respuesta en caché, puede seguir usándola.

4. "no-cache": Verifica primero en el servidor si hay cambios. Si la respuesta no ha cambiado, usa la caché local. Si ha cambiado, la reemplaza con la nueva. Requiere que el servidor envíe encabezados como ETag o Last-Modified.

5. "force-cache": Usa la caché si existe **sin verificar en el servidor**. Si no hay una respuesta en caché, va al servidor. Útil para mejorar el rendimiento si los datos no cambian con frecuencia.

6. "only-if-cached": Solo usa la caché, no va al servidor. Si no hay respuesta en caché, falla con un error 504 (Gateway Timeout). ❗ **Solo funciona con solicitudes GET y en el mismo origen (same-origin)**.

¿Cuándo usar cada uno?

- **default o force-cache**: Para mejorar el rendimiento en datos poco cambiantes.
- **no-cache**: Cuando los datos pueden cambiar, pero quieres evitar descargas innecesarias.
- **reload o no-store**: Para obtener siempre la información más reciente.
- **only-if-cached**: Para evitar descargas si no hay datos almacenados en caché.

- **redirect**:

1. "follow" (Por defecto). Sigue automáticamente las redirecciones. Devuelve la respuesta final después de seguir todas las redirecciones. Es el comportamiento predeterminado y más común.

2. "error": Bloquea redirecciones y lanza un error si la respuesta es 301, 302, etc. Útil si no quieres que la solicitud siga redirecciones automáticamente.

3. "manual": No sigue automáticamente las redirecciones, pero tampoco lanza error. La respuesta tendrá un estado 3xx y una cabecera Location con la URL de redirección. Útil cuando necesitas manejar manualmente las redirecciones.

- **referrerPolicy**:

1. "no-referrer": No envía la cabecera Referer en la solicitud. Útil para privacidad, ya que el servidor no sabe de dónde viene la petición.

2. "no-referrer-when-downgrade" (Por defecto): Envía la cabecera `Referer` solo si la solicitud no es de HTTPS → HTTP. Si la página es HTTPS y la solicitud es HTTP, **no se envía** Referer. En solicitudes dentro del mismo protocolo (HTTP → HTTP o HTTPS → HTTPS), sí se envía.

3. "origin": Envía solo el origen (dominio + protocolo) sin la ruta completa.

4. "origin-when-cross-origin": Para solicitudes same-origin: envía la URL completa como Referer. **Para solicitudes cross-origin:** envía solo el origen (como "origin").

5. "same-origin": Solo envía Referer si la solicitud es dentro del mismo origen. Si la solicitud es a otro dominio, **no se envía Referer**.

6. "strict-origin": Siempre envía solo el origen, pero nunca en una solicitud de HTTPS → HTTP.

7. "strict-origin-when-cross-origin" (Más seguro que el default): Para solicitudes same-origin: envía la URL completa. **Para solicitudes cross-origin:** envía solo el origen. **No envía Referer en solicitudes de HTTPS → HTTP.**

- **integrity**: se usa para verificar que el recurso que estamos descargando no ha sido manipulado, mediante un hash criptográfico. Los valores que puede tomar siguen este formato:

  `<hash-algorithm>-<base64-hash>`

  Donde los algoritmos de hash permitidos son:

  - sha256: Hash SHA-256
  - sha384: Hash SHA-384
  - sha512: Hash SHA-512

  Ejemplo:

```
fetch('https://ejemplo.com/script.js', {
integrity: 'sha384-oqVuAfXRKap7fdgcCY5uykM6+R9GqQ8K/uxy9rx7HNQlGYl1kPzQho1wx4JwY8wC'
});
```

  Si el hash del recurso descargado no coincide con el especificado, el navegador rechazará el recurso y la promesa de fetch será rechazada con un error. Este mecanismo es especialmente útil cuando se cargan recursos de terceros, ya que nos permite asegurarnos de que el contenido no ha sido modificado maliciosamente en el camino.

8.3. Response

En la función que pasamos a then() vamos a recibir un **objeto *Response***. Este objeto contiene la respuesta que hace el servidor y dispone de una serie de propiedades con los valores de esa respuesta.

```
fetch('https://httpbin.org/ip')
.then(function(response) {
```

```
    console.log('response.body =', response.body);
    console.log('response.bodyUsed =', response.bodyUsed);
    console.log('response.headers =', response.headers);
    console.log('response.ok =', response.ok);
    console.log('response.status =', response.status);
    console.log('response.statusText =', response.statusText);
    console.log('response.type =', response.type);
    console.log('response.url =', response.url);
    return response.json();
})
.then(function(data) {
    console.log('data = ', data);     })
.catch(function(err) {
    console.error(err);
});
```

El contenido del body no está disponible directamente en este objeto Response y tenemos que llamar a uno de los métodos disponibles para que nos devuelva una promesa donde recibiremos el valor enviado por el servidor. Los métodos disponibles son:

- `response.text()` para que nos devuelve el contenido en formato texto
- `response.json()` para que lo devuelva como objeto Javascript
- `response.arrayBuffer()` para obtenerlo como ArrayBuffer
- `response.blob()` como valor que podemos manejar con `URL.createObjectURL()`
- `response.formData()` para obtenerlos como `FormData`

Una característica que tenemos que tener en cuenta es que sólo podemos hacer una obtención de body, tras la cual ya no podemos volver a solicitar otra conversión. Para resolver esta situación **el objeto Response tiene el método clone()** que nos permite duplicar el objeto y hacer múltiples gestiones de body.

Es interesante saber que también podemos recibir los datos del server usando **response.blob()**. Un objeto **Blob** representa un objeto tipo fichero de datos planos inmutables. Los Blobs representan datos que no necesariamente se encuentran en un formato nativo de JavaScript. Esto quiere decir que podemos descargar ficheros del servidor de la siguiente manera:

```
fetch('flores.jpg')
.then(function(response) {
    if(response.ok) {
      response.blob()
      .then(function(miBlob) {
        var objectURL = URL.createObjectURL(miBlob);
        miImagen.src = objectURL;
      });
    } else {
      console.log('Respuesta de red OK.');
```

```
    }
})
.catch(function(error) {
  console.log('Hubo un problema con la petición Fetch:' + error.message);
});
```

El método estático URL.createObjectURL() crea un **DOMString** que contiene una URL que representa al objeto pasado como parámetro. La vida de la URL está ligada al **document** de la ventana en la que fue creada. El nuevo objeto URL representa al objeto File especificado o al objeto Blob.

8.4. Objeto Request

Una forma alternativa de configurar el comportamiento de fetch() es crear un objeto Request y pasar este objeto como parámetro a fetch(). El constructor de Request recibe dos parámetros: la URL y el objeto con las opciones.

```
var request = new Request('https://httpbin.org/get', {
    method: 'GET',
    mode: 'cors',
    credentials: 'omit',
    cache: 'only-if-cached',
    referrerPolicy: 'no-referrer'
});

console.log('request =', request);

fetch(request)
.then(function(response) {
    console.log('response =', response);
    return response.text();
})
.then(function(data) {
    console.log('data = ', data);
})
.catch(function(err) {
    console.error(err);
});
```

Este objeto Request puede ser de utilidad si tenemos que hacer varias llamadas con los mismos valores, ya que podemos reutilizarlo tantas veces como nos haga falta. Acepta todas las configuraciones que vimos en fetch().

Si capturamos el evento fetch recibiremos un objeto event.request igual al que hemos utilizado al realizar la llamada. Para acceder al body de este Request debemos utilizar las mismas llamadas que hicimos anteriormente con Response. Realmente Response y Request implementan la interfaz body que dispone de los diferentes métodos para el acceso al contenido del body.

8.5. Objeto Headers

Aunque podemos incluir las cabeceras por medio del objeto Request o como parte del objeto que se pasa como segundo parámetro a fetch(), tenemos a nuestra disposición el objeto Headers que nos ayuda a gestionar las cabeceras de una forma más precisa.

```
var headers = new Headers();

headers.append('a', '1');
headers.append('b', '2');
var request = new Request('https://httpbin.org/get', {
    headers: headers
    });
console.log('request =', request);
for (var k of request.headers.keys()) {
    console.log('request.headers.get("' + k + '") =', request.headers.get(k));
}
fetch(request)
    .then(function(response) {
        console.log('response =', response);
        for (var k of response.headers.keys()) {
            console.log('response.headers.get("' + k + '") =',
response.headers.get(k));
        }
        return response.text();
    })
    .then(function(data) {
        console.log('data = ', data);
    })
    .catch(function(err) {
        console.error(err);
    });
```

El objeto Headers se puede asignar a Request, pero también está presente en Response con las cabeceras que nos devuelve el servidor. Si queremos consultar las cabeceras devueltas tenemos que utilizar los métodos que ofrece Headers:

- `Headers.append(key, value)`: añade un valor a una cabecerá ya existe o crea una nueva cabecera si no existe.
- `Headers.delete(key)`: borra una cabecera.
- `Headers.entries()`: retorna un iterador con todas las parejas clave/valor.
- `Headers.get(key)`: devuelve el primer valor de una cabecera.
- `Headers.getAll(key)`: devuelve una matriz con todos los valores de una cabecera.
- `Headers.has(key)`: comprueba si una cabecera existe.
- `Headers.keys()`: devuelve un interador con todas las claves.
- `Headers.set(key, value)`: añade una cabecera nueva.
- `Headers.values()`: devuelve un iterador con todos los valores.