Tema 4: El objeto XMLHTTPRequest

- El objeto XMLHTTPRequest
- Creación del objeto XMLHTTPRequest
- Uso básico del objeto XMLHTTPRequest
- Métodos y propiedades del objeto XMLHttpRequest
- Refactorización
- Envío de parámetros con la petición HTTP
- Refactorizando la utilidad CargadorContenidos
- Uso de la nueva utilidad
- Envío de parámetros mediante XML
- Procesando respuestas XML

El objeto XMLHTTPRequest

- 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 Web Access.

- Este objeto nos va a permitir realizar llamadas asíncronas al servidor, de forma que podremos cambiar los contenidos del documento, o parte del mismo, sin tener que volver a recargar la página.

Creación del objeto XMLHTTPRequest

- 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

```
if (window.XMLHttpRequest)
        peticion= new XMLHttpRequest();
else if (window.ActiveXObject)
        peticion new ActiveXObject("Microsoft.XMLHTTP");
```

- Los navegadores que siguen los estándares (Firefox, Safari, Opera, Internet Explorer 7 y 8) 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.

Uso básico del objeto XMLHTTPRequest

- Una vez que hemos creado el objeto ya podemos realizar peticiones asíncronas al servidor a través de él.
- Para realizar una petición al servidor, lo primero que debemos hacer es preparar la función de respuesta, es decir, indicarle al objeto cual será la función que se encargue de procesar la respuesta del servidor a la llamada que realicemos.
- Se lo indicaremos a través de la propiedad **onreadystatechange** del objeto:

```
peticion.onreadystatechange=procesaRespuesta
```

- Evidentemente, la función procesaRespuesta tiene que estar definida.

Uso básico del objeto XMLHTTPRequest (II)

- Una vez que le hemos indicado al objeto cual va a ser la función que procesará la respuesta del servidor, ya podemos realizar la llamada:

```
peticion.open('GET', 'http://localhost/prueba.txt');
peticion.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) y la url solicitada (http://localhost/prueba.txt).
- 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 veremos en detalle todos los métodos y propiedades que permiten hacer las peticiones al servidor.

Uso básico del objeto XMLHTTPRequest (III)

- Únicamente nos queda ver como sería la función procesaRespuesta:

```
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;

function procesaRespuesta()
{
   if (peticion.readyState==READY_STATE_COMPLETE)
   {
        if (peticion.status==200)
        {
            var contenido=document.getElementById("contenido");
            contenido.innerHTML="<p>" + peticion.responseText + "</p>";
        }
    }
}
```

Uso básico del objeto XMLHTTPRequest (IV)

- Esta función será invocada automáticamente cada vez que se produzca un cambio en el estado de la petición que hemos realizado.
- Este cambio de estado se reflejará en la propiedad readyState.
- Los estados que puede tomar esta propiedad 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)

- Dado que esta propiedad sólo puede tomar esos valores se suelen definir constantes globales con estos valores para clarificar el código.

Uso básico del objeto XMLHTTPRequest (V)

- La función procesaRespuesta comprobará, 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 comprobará que sea válida y correcta (comprobando si el código de estado http devuelto es igual a 200).
- Una vez realizadas las comprobaciones, simplemente se añade al elemento contenido de la página la respuesta del servidor (en este caso, el contenido del archivo solicitado) mediante la propiedad responseText.

Veamos una serie de ejemplos del uso básico del objeto XMLHttpRequest

Métodos y propiedades del objeto XMLHttpRequest

- El objeto XMLHttpRequest posee muchas otras propiedades y métodos que no hemos visto en el ejemplo anterior.
- A continuación se incluye la lista completa de todas las propiedades y métodos del objeto.

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
`responseXML`	El contenido de la respuesta del servidor en formato XML. El objeto devuelto se puede procesar como un objeto DOM
`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.

Métodos y propiedades del objeto XMLHttpRequest (II)

- 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()

Métodos y propiedades del objeto XMLHttpRequest (III)

- 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.

Métodos y propiedades del objeto XMLHttpRequest (IV)

- 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. Si no se envían datos, se debe indicar un valor igual a null.
- En otro caso, se puede indicar como parámetro una cadena de texto, un array de bytes o un objeto XML DOM.

Refactorización

- Como hemos visto en el ejemplo anterior es muy sencillo realizar una petición Ajax al servidor, el problema viene cuando nuestra aplicación crece y tenemos que realizar una gran cantidad de peticiones.
- En estos casos puede resultar muy tedioso y propenso a errores repetir el código anterior para cada una de las peticiones que realicemos. Además, cuanto más crezca nuestra aplicación más difícil nos resultará mantenerla, es decir realizar modificaciones en la misma.
- En este punto veremos un conjunto de herramientas y técnicas que podemos usar para mantener nuestro código limpio, es decir, sencillo de implementar, de mantener y menos propenso a errores.
- La principal herramienta que vamos a utilizar es la refactorización del código, es decir, el proceso de reescribir código para obtener mayor claridad en el mismo sin añadir nuevas funcionalidades.

Refactorización (II)

- Cuando el archivo javascript de nuestra aplicación crece en exceso (120 líneas o más), y se mezclan en este, funcionalidades de bajo nivel, como podría ser la interacción con el objeto XMLHTTPRequest, con otras funcionalidades de más alto nivel, como por ejemplo, el tratamiento de las respuestas a la petición Ajax, se hace necesario separar este archivo en distintos archivos javascript que sean más pequeños, más fáciles de mantener, y cuyo código interno tenga una funcionalidad común.

- Esto a menudo se conoce como *"separación de responsabilidades"*.

Refactorización (III)

- Veamos como podríamos refactorizar el ejemplo anterior para clarificar el código y facilitar su reutilización.
- El ejemplo anterior tiene el problema de no permitir la reutilización de código.
  - ✓ en primer lugar porque se utiliza una variable global para contener el objeto XMLHTTPRequest creado.
  - ✓ en segundo lugar porque el propio código que crea el objeto y realiza la petición contiene también la gestión de la respuesta.
- Esto puede ser problemático en el caso que queramos realizar varias peticiones simultáneas al servidor, dado que tendríamos que crear diferentes manejadores para cada petición, es decir,
  - ✓ tendríamos que repetir el código del ejemplo para cada una de las peticiones que realicemos.

Refactorización (IV)

- En programación orientada a objetos, la solución a este tipo de problemas es encapsular la funcionalidad requerida en un objeto que pueda ser reutilizado.
- Veremos a continuación, cómo encapsular todo este proceso en un objeto que llamaremos **CargadorContenidos**, dado que nos servirá para cargar contenidos del servidor.
- En javascript no disponemos de los namespaces, un mecanismo muy utilizado en otros lenguajes de programación para evitar colisiones de nombres entre librerías de clases, pero podemos simularlo introduciendo nuestros objetos de utilidad en otro objeto que haga de namespace.
- Así, si introducimos nuestro objeto **CargadorContenidos** en otro objeto **miAjax** que contenga todos los objetos referentes a la gestión de peticiones Ajax, evitaremos estas posibles colisiones de nombres con otras librerías.

Refactorización (V)

❑ Este objeto contendrá también las constantes que utilizaremos en la librería:

```
var miAjax=new Object();
miAjax.READY_STATE_UNINITIALIZED=0;
miAjax.READY_STATE_LOADING=1;
miAjax.READY_STATE_LOADED=2;
miAjax.READY_STATE_INTERACTIVE=3;
miAjax.READY_STATE_COMPLETE=4;
```

❑ La función constructora del objeto CargadorContenidos recibirá los siguientes parámetros:

- ✓ **url**: contendrá la url de la petición que se realizará al servidor.
- ✓ **procesaRespuesta**: será una función que se llamará si la petición se realiza correctamente y se encargará de procesar la respuesta recibida.
- ✓ **procesaError**: será una función que se llamará si se produce un error en la petición y se encargará de procesar dicho error (este parámetro será opcional porque crearemos un manejador de error por defecto).

Refactorización (VI)

- Esta función constructora quedará de la siguiente forma:

```
miAjax.CargadorContenidos=function(url, procesaRespuesta, procesaError){
    this.url=url;
    this.procesaRespuesta=procesaRespuesta;
    this.procesaError=(procesaError) ? procesaError : this.defaultError;
    this.peticion=this.creaPeticion();
    this.cargaContenidos(url);
}
```

Refactorización (VII)

- A continuación, definiremos un método (creaPeticion) que se encargará de crear el objeto XMLHTTPRequest (obsérvese que este método es llamado desde la función constructora):

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

Refactorización (VIII)

- El siguiente método (cargaContenidos) será el encargado de realizar la petición al servidor y llamar a la función que procesará la respuesta si todo va bien, o a la función que procesa el error si este se produjera (obsérvese que este método es llamado desde la función constructora):

```
cargaContenidos:function(url)
  {
    if (this.peticion)
      {
         try
         {
            var loader = this;
            this.peticion.onreadystatechange=function()
            {
               loader.onReadyState.call(loader);
            }
            this.peticion.open('GET',url,true);
            this.peticion.send(null);
         }catch (err)
         {
            this.procesaError.call(this);
         }
      }
  }
```

Refactorización (IX)

- El método call de javascript permite ejecutar una función como si fuera un método de un objeto pasándole el propio objeto como parámetro, de esta forma desde la función se podrá acceder al objeto a través de la variable this.

- La variable loader es necesaria para poder acceder al objeto this dentro de la función que se le asigna al evento onreadystatechange (en javascript cuando se define una función dentro de otra, todas las variables de la primera función están disponibles de forma directa en la función anidada).

Refactorización (X)

- Lo siguiente que tendremos que definir es el método al que se llama para verificar el estado de la petición http (onReadyState).
- Este método será llamado cada vez que se produzca un cambio de estado en la petición y será el encargado de llamar a las funciones procesaRespuesta o procesaError, según el caso:

```
onReadyState:function()
  {
     if (this.peticion.readyState==miAjax.READY_STATE_COMPLETE)
     {
        if (this.peticion.status==200 || this.peticion.status==0)
        {
           this.procesaRespuesta.call(this);
        }
        else
        {
           this.procesaError.call(this);
        }
     }
  }
```

Refactorización (XI)

- Por último, sólo nos quedará definir la función de error por defecto que se llamará si no hemos pasado una función específica en la creación del objeto:

```
defaultError:function()
  {
     alert("Se ha producido un error en la petición Ajax al servidor!"
           +"\n\nreadyState:"+this.peticion.readyState
           +"\nstatus: "+this.peticion.status
           +"\nheaders: "+this.peticion.getAllResponseHeaders());
  }
```

- El ejemplo completo quedaría de la siguiente forma (nótese que el objeto está definido con notación JSON y utilizando la propiedad prototype para definir el prototipo del mismo):

Refactorización (XII)

- Un ejemplo de creación del objeto será el siguiente:

```
var cargador= new miajax.CargadorContenidos("pagina.php",procesaDatos)
```

- En la llamada, el parámetro "procesaDatos" será el nombre de la función que se encargará de procesar los datos devueltos por el servidor.

```
function procesaDatos ()
{
  alert (this.url
        + "Respuesta cargada. Estos son los datos \n\n"
        + this.peticion.reponseText)
}
```

- Si necesitamos acceder a las propiedades del objeto cargador, podremos hacerlo a través de la variable this, ya que es llamada utilizando el método call.

Con esta refactorización hemos conseguido tener un objeto que internamente es complejo, pero que es muy sencillo de utilizar externamente.

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 este objeto son muy superiores, ya que también permite el envío de parámetros junto con la petición http.
- Se pueden 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
```

Envío de parámetros con la petición HTTP (II)

- 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.

Envío de parámetros con la petición HTTP (III)

- 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 si que va a enviar los parámetros, por lo que recibirá la cadena construida a partir de los datos del formulario.
- Una buena práctica muy utilizada para evitar problemas con la caché del navegador es añadir al final de la cadena un parámetro llamado nocache que contiene un número aleatorio (creado mediante el método Math.random()).
- 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 caché.

Envío de parámetros con la petición HTTP (IV)

- Una vez que hemos construido la cadena separando los parámetros por '&' hemos de asegurarnos de que en el interior de la cadena no aparece ningún carácter especial que no pueda ser usado directamente en una url.
- Para ello utilizaremos la función encodeURIComponent() que 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.
- Las sustituciones más conocidas son las de los espacios en blanco por %20, y la del símbolo & por %26, pero también se sustituyen todos los acentos y cualquier otro carácter que no se puede incluir directamente en una url.
- Javascript incluye una función contraria llamada decodeURIComponent() y que realiza la transformación inversa.

Envío de parámetros con la petición HTTP (V)

❑ Una vez que tenemos la cadena construida, se la pasaremos como parámetro al método send del objeto XMLHttpRequest, pero antes debemos tener en cuenta una serie de cosas:

✓ En el método open del objeto XMLHttpRequest le pasamos como primer parámetro el método (GET / POST) que utilizaremos en la llamada al servidor. Normalmente cuando se trata de datos introducidos por el usuario en un formulario se utiliza el método POST debido a las limitaciones del método GET comentadas anteriormente.

✓ 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.setRequestHeader( "Content-Type", “application/x-www-form-urlencoded”)
```

Envío de parámetros con la petición HTTP (VI)

- Con todo esto, el código que realizaría la llamada al servidor sería el siguiente:

```
peticion.open(“POST”, “pagina.php”,true);
peticion.setRequestHeader( "Content-Type", “application/x-www-form-urlencoded”);
var query_string=crea_query_string();
peticion.send(query_string);
```

Refactorizando la utilidad CargadorContenidos

- La utilidad diseñada anteriormente para la carga de contenidos y recursos almacenados en el servidor, solamente está preparada para realizar peticiones HTTP sencillas mediante GET.
- A continuación se refactoriza esa utilidad para que permita las peticiones POST y el envío de parámetros al servidor.
- El primer cambio necesario es el de adaptar el constructor para que se puedan especificar los nuevos parámetros:

```
miAjax.CargadorContenidos=function(
                    url, procesaRespuesta, procesaError, metodo, parametros, contentType)
```

- Se han añadido tres nuevos parámetros: el método http empleado, los parámetros que se envían al servidor junto con la petición y el valor de la cabecera content-type.

Refactorizando la utilidad CargadorContenidos (II)

- Estos parámetros habrá que pasárselos a la función que se encarga de realizar la petición (cargaContenidos):

```
this.cargaContenidos(url, metodo, parametros, contentType);
```

- En esta función se utilizan estos parámetros para abrir la petición antes de enviarla (open):

```
this.peticion.open(metodo,url,true);
```

- El siguiente paso es añadir (si así se indica) la cabecera Content-Type de la petición:

```
if (contentType)
        {
            this.peticion.setRequestHeader("Content-Type", contentType);
        }
```

Refactorizando la utilidad CargadorContenidos (III)

- Por último, se añade la cadena de parámetros a la función send:

```
this.peticion.send(parametros);
```

- De esta forma hemos implementado un objeto que nos permitirá realizar peticiones AJAX con los dos metodos disponibles GET y POST, permitiéndonos pasar información en la peticion.