Clase 4

Ajax

XML
Ajax

definición
Breve explicación de como funciona el HTTP
XMLHttpRequest

XML

El XML se creó para que cumpliera varios objetivos.

- Que fuera idéntico a la hora de servir, recibir y procesar la información que el HTML, para aprovechar toda la tecnología implantada para este último.
- Que fuera formal y conciso desde el punto de vista de los datos y la manera de guardarlos.
- Que fuera extensible, para que lo puedan utilizar en todos los campos del conocimiento.
- Que fuese fácil de leer y editar.
- Que fuese fácil de implantar, programar y aplicar a los distintos sistemas.

El XML se puede usar para infinidad de trabajos y aporta muchas ventajas en amplios escenarios. Veamos algunas ventajas del XML en algunos campos prácticos.

- Comunicación de datos. Si la información se transfiere en XML, cualquier aplicación podría escribir un documento de texto plano con los datos que estaba manejando en formato XML y otra aplicación recibir esta información y trabajar con ella.
- Migración de datos. Si tenemos que mover los datos de una base de datos a otra sería muy sencillo si las dos trabajasen en formato XML.
- Aplicaciones web. Hasta ahora cada navegador interpreta la información a su manera y los programadores del web tenemos que hacer unas cosas u otras en función del navegador del usuario. Con XML tenemos una sola aplicación que maneja los datos y para cada navegador o soporte podremos tener una hoja de estilo o similar para aplicarle el estilo adecuado. Si mañana nuestra aplicación debe correr en WAP solo tenemos que crear una nueva hoja de estilo o similar.

Son sólo unos ejemplos que esperamos que comprendas aunque sea por encima ya que todavía hay muchas cosas que no sabes sobre XML y las tecnologías relacionadas.

Ajax

Definición

Ajax, acrónimo de ***A**synchronous **J**avaScript **A**nd **X**ML* (JavaScript asíncrono y XML), es una técnica de desarrollo web para crear aplicaciones interactivas o RIA (*Rich Internet Applications*). Estas aplicaciones se ejecutan en el cliente, es decir, en el navegador de los usuarios mientras se mantiene la comunicación asíncrona con el servidor en segundo plano. De esta forma es posible realizar cambios sobre las páginas sin necesidad de recargarlas, lo que significa aumentar la interactividad, velocidad y usabilidad en las aplicaciones.
(wikipedia)

Breve explicación de como funciona HTTP

Recordemos, ante que nada, que la WWW funciona con un modelo de cliente-servidor, siendo el cliente el navegador (Firefox, Mozilla, Opera o el engendro) y el servidor un demonio http (Apache, Zeus y el otro engendro).
En una carga de página normal, el cliente envía lo que se llama una petición al servidor. Cuando ponemos en la barra del navegador "http://php.apsique.com", enviamos el siguiente texto al servidor por un puerto, que generalmente es el 80:
GET / HTTP/1.0
Como vemos en la especificación del protocolo HTTP, existen diversos métodos a través de los cuales podemos decir al servidor que tipo de petición debemos hacer. Tenemos los clásicos GET Y POST, pero también otros como HEAD, DELETE y PUT. Lo importante es recordar que GET sirve para recuperar información sin alterar la fuente, en tanto que POST nos permite entregar información que altera el recurso.
El servidor, al recibir una petición, nos entrega una respuesta, que es también un texto. Esta respuesta siempre comienza con un código, que indica el estado de la operación. Entre ellos tenemos el esperado 200 (todo ok), 301 (movido permanentemente) y los temidos 403 (prohibido), 404 (no existe recurso) y 500 (error del cgi). Después de esta respuesta, generalmente se ofrece el tipo MIME del contenido (texto, xml, imagen) y el contenido propiamente tal. Por ejemplo, para una página típica de mi sitio

```
Date: Mon, 30 May 2005 07:12:49 GMT
Server: Apache/1.3.33 (Unix)  (Gentoo/Linux) mod_ruby/1.2.4 Ruby/
1.8.2(2004-12-25) PHP/4.3.11
X-Powered-By: PHP/4.3.11
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Type: text/html; charset=utf-8
```

200 OK

Si bien en una carga de página normal todo este enredo lo manejan de forma casi mágica el cliente y el servidor de forma automática, al ocupar Ajax hay que manejar todo esto, en el fondo, *a mano*. Si bien ya existen bastantes aplicaciones que ocultan esta información, si no tienen claro el proceso de consulta y respuesta, no van a poder entender bien como funciona el proceso en realidad y de donde surgen los errores.

XMLHTTPRequest

Ya teniendo en nuestras cabezas el proceso de consulta-respuesta, podemos empezar a trabajar en Ajax. Si quieren un texto muy bueno, y en el cual se basará la discusión, lean Using the XML HTTP Request object
El nombre Ajax es un acrónimo de "Asynchronous JavaScript + XML". Ahora, *no es necesario ocupar XML*. Si, tal como lo oyeron. Si bien siempre suena bonito ocupar esta siglita, la verdad es que la consulta y recepción son simples textos; así, si bien se puede ocupar XML para ordenar la cosa, no estamos limitados por ello.

Javascript al mando

A partir del Explorer 6 y de Mozilla 1.6, tenemos a nuestra disposición el objeto XMLHttpRequest. El problemita es que en IE (para variar) se accede a el con
new ActiveXObject("Msxml2.XMLHTTP")
o
new ActiveXObject("Microsoft.XMLHTTP")
dependiendo de la versión del IE. Por tanto, lo mejor es crear una función que nos entregue el objeto, la cual diría algo como:

```
function get_xmlhttp() {
try {
 xmlhttp = new ActiveXObject("Msxml2.XMLHTTP");
 } catch (e) {
 try {
 xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
 } catch (E) {
 xmlhttp = false;
 }
 }
if (!xmlhttp && typeof XMLHttpRequest!='undefined') {
 xmlhttp = new XMLHttpRequest();
}
return xmlhttp
}
```

Una vez que tenemos el objeto, para hacer una petición debemos abrir la conexión y enviar la consulta. Para ello utilizados

1. xmlhttp.open(metodo, url,async);
2. xmlhttp.send(contenido);

Recuerden que si utilizamos "GET", bastará con poner las variables en el url, en el clásico formato
script.php?variable1=valor1&variable2=valor2
Escapando los carácteres extraños con *encodeURIComponent*. Si ocupan POST, deben enviar el contenido en la variable *contenido* en *send*, aunque por ahora no sé muy bien como se hace (apenas tenga el dato, aviso)
Normalmente, nosotros deseamos saber en que momento el servidor nos entrega la respuesta y realizar una función con ésta. Para ello, existe una función de xmlhttp llamada *onreadystatechange*, la cual se ejecuta cada vez que cambia el estado de la conexión. Sabiendo que la propiedad *readyState* se pone a *4* cuando la respuesta ha sido recibida, podemos mejorar el código haciendo

```
xmlhttp.open(metodo, url,async);
  xmlhttp.onreadystatechange=function() {
  if (xmlhttp.readyState==4) {
  callback(xmlhttp)
  }
  }
  xmlhttp.send(null)
```

callback es el nombre de una función que recibirá el resultado de la llamada al servidor
Si revisamos el objeto xmlhttp una vez recibida la respuesta, veremos que cuenta con los siguientes métodos de interés
getAllResponseHeaders()
Nos entrega todas las cabeceras de respuesta desde el servidor
status
El código de estatus de la petición (normalmente, 200)
responseText
Texto literal de la respuesta
responseXML
Objeto XMLDocument, el cual se puede analizar con las funciones de DOM típicas, como getElementById y similares.De aquí la *X* de Ajax

Realizar la petición al servidor

Para realizar la petición al servidor, utilizaremos los métodos **open**, **onreadystatechange** y **send**, que sirven respectivamente para preparar la petición, seleccionar la función de recepción e iniciar la petición.

Al método open, hay que pasarle el método de petición (GET) y la URL que se enviará al servidor y mediante la cual, el servidor, creará la respuesta que posteriormente leeremos.

Para nuestro primer ejemplo vamos a pedir un documento de texto:

```
<script>
// Creamos el objeto
oXML =AJAXCrearObjeto();
// Preparamos la petición
oXML.open('GET', 'archivo.txt');
// Preparamos la recepción
oXML.onreadystatechange = leerDatos;
// Realizamos la petición
oXML.send('');
</script>
```

Para que esto funcione, tendremos que haber declarado la función **leerDatos** para tratar los datos recibidos del servidor y mostrarlos al usuario, pero esto lo veremos más adelante.

Paso de parámetros

En la petición AJAX podemos pasar parámetros tanto por POST como por GET a nuestro servidor.

Para pasar parámetros por GET (por URL) , usaremos una URL con parametros en la función **open** independientemente de usar el método GET o POST, por ejemplo:

```
<script>
// Creamos la variable parametro
parametro = 'Datos pasados por GET';
// Creamos el objeto
oXML = AJAXCrearObjeto();
// Preparamos la petición con parametros
oXML.open('GET', 'pagina.php?parametro=' + escape(parametro));
// Preparamos la recepción
oXML.onreadystatechange = leerDatos;
// Realizamos la petición
oXML.send('');
</script>
```

Para pasarlos por POST, deberemos usar el método POST en la función **open** y pasar los parámetros desde la función **send**, veamos un ejemplo:

```
<script>
// Creamos la variable parametro
```

```
parametro = 'Datos pasados por POST';
// Creamos el objeto
oXML = AJAXCrearObjeto();
// Preparamos la petición con parametros
oXML.open('POST','pagina.php');
// Preparamos la recepción
oXML.onreadystatechange = leerDatos;
// Realizamos la petición
oXML.send( 'parametro=' + escape(parametro));
</script>
```