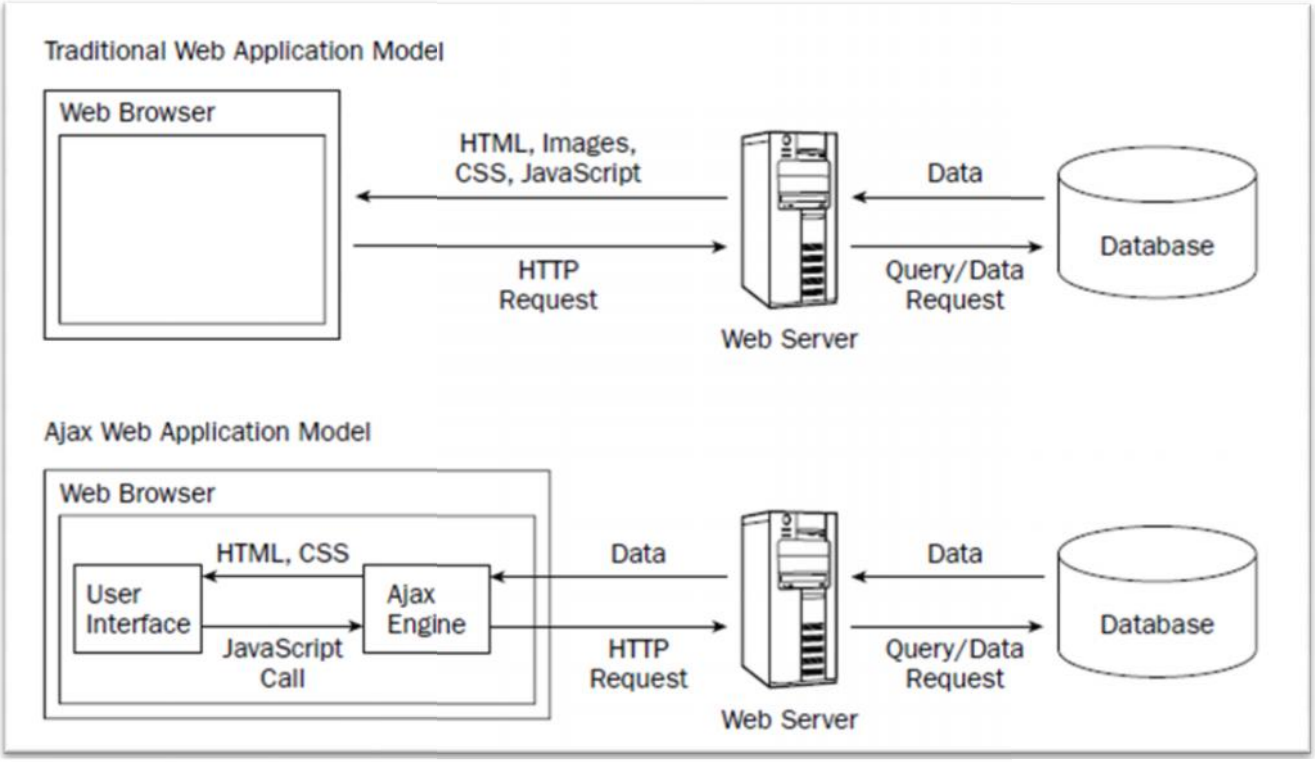
AJAX

La evolución de la World Wide Web, en términos de demanda de servicios por parte de los usuarios, ha generado nuevos paradigmas de interactividad que han propiciado la inclusión de nuevas funcionalidades en los navegadores (clientes). Como resultado, se ha ido variando progresivamente la lógica en el desarrollo de las aplicaciones web. Más específicamente hablamos de la carga dinámica de fragmentos de páginas mediante la transmisión de pequeñas cantidades de datos del servidor al cliente, de manera que la percepción del usuario sea cada vez más cercana a la utilización de una aplicación de escritorio.

La primera técnica que se desarrolló con amplia compatibilidad dentro de los distintos navegadores para crear aplicaciones web interactivas es AJAX, y que sigue siendo la base conceptual de cómo se realizan todo este tipo de interacciones de forma más dinámica entre cliente y servidor en la web.

AJAX no consiste en una tecnología específica, ni es un lenguaje de programación determinado. El término AJAX hace referencia a la utilización de peticiones HTTP asíncronas empleando por lo general Javascript con el propósito de recibir información del servidor sin necesidad de recargar la página web mostrada al usuario. Las peticiones se pueden realizar de distintas maneras, y de igual forma se pueden emplear distintos formatos para los datos intercambiados.

Para entender mejor el concepto, podemos comparar el modelo clásico con el basado en AJAX:



En el modelo clásico, el usuario interacciona con el sistema mediante las opciones que pone a su disposición la interfaz mostrada en su navegador. Cualquier acción realizada por el usuario desemboca en una petición al servidor Web, que en ocasiones se apoya en una base de datos para recuperar los datos requeridos por el navegador. Como resultado se devuelve una nueva página HTML que es visualizada en el cliente.

Por el contrario, en el modelo basado en AJAX existe una nueva capa entre la interfaz de usuario y el servidor, encargada de realizar las peticiones HTTP de los datos requeridos para la actualización de la página mostrada al usuario, pero sin que sea necesario volver a cargar toda ella (con los elementos comunes como imágenes, anuncios, etc. que posiblemente ralentizarían la carga).

El objeto XMLHttpRequest

XMLHttpRequest (XHR) es una interfaz empleada para realizar peticiones HTTP y HTTPS a servidores Web. Para los datos transferidos se usa cualquier codificación basada en texto, incluyendo texto plano, XML, JSON, HTML y codificaciones particulares específicas. La interfaz se implementa como una clase de la que una aplicación cliente puede generar tantas instancias como necesite para manejar el diálogo con el servidor.

El objeto *XMLHttpRequest* constituye una parte fundamental de las aplicaciones AJAX, y fue introducido en los navegadores con el fin de poder iniciar peticiones HTTP de forma asíncrona desde cualquier parte de una aplicación.

Mediante una librería ActiveX llamada MSXML. Posteriormente otros navegadores como Firefox, Safari u Opera implementaron la misma funcionalidad, hasta que en 2006 el W3C presentó la primera versión para una especificación estándar de la interfaz. En el caso de IE debemos utilizar la clase propietaria ActiveXObject y utilizar la firma del control correspondiente. Es deseable utilizar la versión más reciente de la librería MSXML disponible en el navegador. Las firmas de este componente para distintas versiones se muestran a continuación:

- Microsoft.XMLHttp
- MSXML2.XMLHttp
- MSXML2.XMLHttp.3.0
- MSXML2.XMLHttp.4.0
- MSXML2.XMLHttp.5.0

Algunos ejemplos de instanciaciones son:

```
var obj=new ActiveXObject(“Microsoft.XMLhttp”);
var obj=new ActiveXObject(“MSXML2.XMLHttp.4.0”);
```

Ejemplo de creacion de objeto *XMLHTTPRequest*:

```
function getXMLHTTPRequest(){
 try{
 req = new XMLHttpRequest();
 } catch(err1) {
   try{
   req = new ActiveXObject("MSXML2.XMLHttp.5.0");
   } catch (err2) {
      try {
      req = new ActiveXObject("MSXML2.XMLHttp.4.0");
      } catch (err3) {
         try {
         req = new ActiveXObject("MSXML2.XMLHttp.3.0");
         } catch (err4) {
            try {
            req = new ActiveXObject("MSXML2.XMLHttp");
            } catch (err5) {
               try {
               req = new ActiveXObject("Microsoft.XMLHttp");
               } catch (err6) {
                  req = false;
               }
            }
         }
     }
   }
}
return req;
}

function getXMLHTTPRequest(){
versiones = ["MSXML2.XMLHttp.5.0", "MSXML2.XMLHttp.4.0",
             "MSXML2.XMLHttp.3.0", "MSXML2.XMLHttp",
             "Microsoft.XMLHttp"];
if(typeof XMLHttpRequest != "undefined"){
   return new XMLHttpRequest();
} else {
   for (var i = 0; i < versiones.length; i++) {
      try{
      req = new ActiveXObject(versiones[i]);
      return req;
      } catch (err1) {
         // Esto evita que se genere un error y pare la ejec.
      }
   }
}
}
```

Utilizando cualquiera de las funciones mostradas, podemos crear el objeto XMLHttpRequest mediante una llamada del tipo:

```
var http = getXMLHTTPRequest();
```

Como ya hemos indicado, se trata de un objeto que podremos instanciar en nuestro código Javascript haciendo uso de la sentencia new. Este objeto tiene las siguientes propiedades:

- **onreadystatechange**: determina el manejador de eventos que será llamado cuando la propiedad readyState cambie
- **readystate**: Identificador de estado de la petición (0-No iniciada, 1-Cargando, 2-Cargado 3-Interactivo, 4-Completado).
- **responseText**: Datos devueltos en forma de cadena de texto
- **responseXML**: Datos devueltos en forma de objeto XML
- **status**: Código de estado HTTP devuelto por el servidor
- **statusText**: "Phrase reason" HTTP devuelta por el servidor.

Además de estas propiedades, cuenta con los siguientes métodos:

- **abort()**: Detiene la petición actual
- **getAllResponseHeaders()**: Devuelve una cadena con todas las cabeceras
- **getresponseHeader()**: Devuelve el valor de la cabecera indicada como una cadena
- **open()**: Especifica el método http, la URL objetivo, y si la petición debe ser asíncrona o síncrona.
- **send()** : Envía la petición
- **setRequestHeader('x','y')**: Configura un par parámetro-valor y lo añade a las cabeceras para ser enviado con la petición

Inicialización del objeto

Después de crear un objeto XMLHttpRequest procedemos a realizar peticiones HTTP desde Javascript. Para ello debemos llamar en primer lugar al método open(), que se encarga de inicializar el objeto.

Los argumentos aceptados por open() son los mostrados en la tabla anterior. Debemos tener en cuenta que el tercer argumento es muy importante, ya que controla cómo ejecuta Javascript la petición. Si tiene un valor *true* la petición se envía de forma asíncrona y el código Javascript se continúa ejecutando sin esperar la respuesta. Si el argumento es *false*, la petición se envía de forma síncrona y se espera a una respuesta del servidor antes de continuar.

```
var http = etXMLHTTPRequest();
var url = "midocumento.txt";
http.open("GET", url, true);
```

El siguiente paso consiste en definir el manejador de eventos onreadystatechange. El objeto XMLHttpRequest incluye una propiedad llamada readystate que cambia cuando

se realiza una petición y se recibe la respuesta. Los cinco valores posibles se muestran en la tabla correspondiente, y se comentan más en detalle a continuación:

- 0 – Sin inicializar – El objeto se ha creado pero no se ha llamado al método open() .
- 1 – Cargando – Se ha llamado al método open() pero la petición todavía no se ha enviado.
- 2 – Cargado – La petición se ha enviado
- 3 – Se ha recibido una respuesta parcial
- 4 – Completo – Se han recibido los datos y se ha cerrado la conexión

Cuando se produzca un cambio en la propiedad `readyState`, de acuerdo con los valores anteriormente mostrados, será invocado el método que indiquemos en `onreadystatechange`. En el ejemplo anterior se ha incluido esquemáticamente el código Javascript que deseamos ejecutar, pero también podemos asignar el nombre de una función que hayamos definido.

```
var http = getXMLHTTPRequest();
var url = "midocumento.txt";
http.open("GET", url, true);
http.onreadystatechange = function () {
     if (http.readystate == 4) {
     // Se ha recibido la respuesta.
     ……..
     }
}
```

Envío de la petición HTTP

El siguiente paso consiste en enviar la petición utilizando el método send(), que recibe como argumento la cadena con el cuerpo de la petición. Debemos recordar que en ocasiones el cuerpo de la petición está vacío (por ejemplo, en las peticiones GET). En estos casos se pasa el valor null.

```
var http = getXMLHTTPRequest();
var url = "midocumento.txt";
http.open("GET", url, true);
http.onreadystatechange = function () {
     if (http.readystate == 4) {
     // Se ha recibido la respuesta.
     ……..
     }
}
http.send(null);
```

Es importante destacar aquí que si enviamos peticiones de forma síncrona no necesitamos asignar el manejador de eventos `onreadystatechange`, dado que en ese caso la respuesta se recibirá cuando finalice el método send(), con lo que después de enviar la petición podemos considerar que ya se ha recibido la respuesta dentro del

script. No obstante, en la gran mayoría de las veces es deseable trabajar con peticiones de manera asíncrona.

Recuperación de la respuesta del servidor

Una vez enviada la petición se seguirá ejecutando el código de la página en el caso de haber enviado la petición de forma asíncrona. De lo contrario se esperará a la respuesta del servidor antes de seguir con la ejecución del script.
Para recuperar los datos devueltos por el servidor podemos utilizar las propiedades `responseText` o `responseXML`.

- `responseText` devuelve una cadena conteniendo el cuerpo de la respuesta.
- `responseXML` devuelve los datos adaptados a un documento XML.

En el ejemplo anterior se ha utilizado una petición GET al servidor con el fin de recuperar un documento de texto. Por tanto deberíamos utilizar la propiedad `responseText`.

Es necesario realizar una comprobación de errores sobre los datos recibidos antes de utilizar la información. La propiedad `status` contiene el código de estado HTTP que se envía en la respuesta, mientras que `statusText` contiene el texto descriptivo del estado. De esta manera podemos comprobar si todo ha salido como esperábamos, o indicarle al usuario la razón de que no se haya completado la operación en caso de fallo.

Utilizando el ejemplo anterior tendremos:

```
var http = getXMLHTTPRequest();
var url = "midocumento.txt";
http.open("GET", url, true);
http.onreadystatechange = function () {
     if (http.readystate == 4) {
          // Se ha recibido la respuesta.
          if(http.status==200) {
               // Aquí escribiremos lo que queremos que
               // se ejecute tras recibir la respuesta
               var datosDoc = http.responseText;
               alert (datosDoc);
          } else {
               // Ha ocurrido un error
               alert("Error: "+http.statusText);
          }
     }
};
http.send(null);
```

Consideraciones sobre el control de la caché del navegador

Con el fin de mejorar la eficiencia al mostrar las páginas Web, los navegadores mantienen un registro local de páginas Web visitadas (caché). Al volver a visitar una página ya vista, el navegador puede emplear la página almacenada de forma local en lugar de volver a consultar al servidor.

Esto es una ventaja en muchas situaciones, pero en el caso de aplicaciones AJAX puede generar efectos no deseados. En muchos casos queremos mantener una conversación continua con el servidor, no con la información almacenada localmente en nuestro navegador.

Existen varias técnicas para evitar el acceso a la caché del navegador cuando programamos una aplicación Web. Dos de las más utilizadas se listan a continuación:

- Asegurarnos de que se llama a una página no almacenada en la cache del navegador.

- Incluir una cabecera en cualquier petición que enviemos al servidor, especificando que no queremos que acceda a la cache para responder.

Un ejemplo utilizando un valor aleatorio:

```
// Hemos creado previamente un objeto
// XMLHttp llamado http
var url = "miurl.php";
var aleat = parseInt(Math.random()*9999999999);
var url_nueva = url+"?aleat="+aleat;
http.open("GET",url_nueva,true);
```

Detener una petición ya realizada

El método abort() del objeto XMLHttpRequest nos permite detener la conexión establecida.

Esto es útil en determinadas situaciones, como cuando el servidor se encuentra saturado y no puede devolver una petición. En estas situaciones es bueno mostrar un mensaje al usuario indicando que existe un problema de sobrecarga en el servidor, y además mediante este método cancelamos la petición previamente realizada.

En estos casos podemos utilizar la función setTimeout() de Javascript, que ejecuta una función determinada al pasar un cierto intervalo de tiempo. Los argumentos que recibe son la función que será ejecutada y el intervalo de tiempo, expresado en milisegundos.

En ocasiones puede ser interesante utilizar setInterval(), similar a setTimeout pero que se ejecuta de forma periódica.

```
function inicializar(url){
     http=crearXMLHttpRequest();
     http.onreadystatechange = procesarEvento;
     http.open("GET", url, true);
     http.send(null);
     tiempo=setTimeout("finDeEspera()",3000);
}
function procesarEvento(){
     var detalles = document.getElementById("detalles");
     if(conexion1.readyState == 4){
          if(http.status==200) {
               clearTimeout(tiempo);
               // Aquí escribiremos lo que queremos que se
               //ejecute tras recibir la respuesta
          } else {
               // Ha ocurrido un error
               alert("Error: "+http.statusText);
          }
     }
}
function finDeEspera(){
     http.abort();
     // Mostrar mensaje de sobrecarga del servidor
     // o en la pagina HTML
     alert('Intente nuevamente más tarde');
}
```