

Unidad 7:

Programación AJAX en JavaScript.


Contenido

1.- Introducción a AJAX.	3
1.1.- Requerimientos previos.	4
1.2.- Comunicación asíncrona.	4
1.3.- El API XMLHttpRequest.	6
1.3.1.- Creación del objeto XMLHttpRequest.	7
1.3.2.- Métodos del objeto XMLHttpRequest.	8
1.3.3.- Propiedades del objeto XMLHttpRequest.	9
2.- Envío y recepción de datos de forma asíncrona.	11
2.1.- Estados de una solicitud asíncrona (parte I).....	12
2.2.- Estados de una solicitud asíncrona (parte II).	13
2.3.- Envío de datos usando método GET.	14
2.4.- Envío de datos usando método POST.	15
2.5.- Recepción de datos en formato XML.	16
2.6.- Recepción de datos en formato JSON (parte I).	17
2.7.- Recepción de datos en formato JSON (parte II).	19
3.- Librerías cross-browser para programación AJAX.....	20
3.1.- Introducción a jQuery (parte I).	22
3.2.- Introducción a jQuery (parte II).	23
3.3.- Introducción a jQuery (parte III).	24
3.4.- Introducción a jQuery (parte IV).	26
3.5.- Función \$.ajax() en jQuery.	27
3.6.- El método .load() y las funciones \$.post() , \$.get() y \$.getJSON() en jQuery.	29
3.7.- Herramientas adicionales en programación AJAX.	31
3.8.- Plugins jQuery.	31

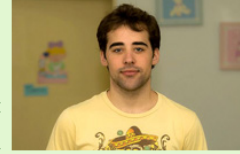


Caso práctico

En estos últimos meses, **Antonio** ha realizado un montón de trabajos en el proyecto, y prácticamente ha terminado todas las tareas. **Juan** le dice que todo el trabajo realizado está muy bien, pero que tendría que actualizar algunos de los procesos para darle un toque de modernidad a la aplicación.

Entre las mejoras que le recomienda Juan, están la de utilizar efectos, más dinamismo, usar AJAX (JavaScript  asíncrono y XML) en las validaciones de los formularios, o en cierto tipo de consultas, etc. El término AJAX le suena muy complicado a **Antonio**, pero **Juan** lo convence rápidamente para que intente hacerlo, ya que no necesita aprender ningún lenguaje nuevo. Utilizando un nuevo objeto de JavaScript, con sus propiedades y métodos va a poder emplear AJAX en sus aplicaciones actuales.

Además, **Juan** lo anima a que se ponga a estudiar rápido el tema de AJAX, ya que al final, le va a dar una pequeña sorpresa, con una librería que le va a facilitar enormemente el programar con AJAX y conseguir dar buenos efectos y mayor dinamismo al proyecto web. Esa librería gratuita tiene el respaldo de grandes compañías a nivel mundial, que la están utilizando actualmente. También cuenta con infinidad de complementos, para que pueda modernizar su web todo lo que quiera, y todo ello programando muy pocas líneas de código y en un tiempo de desarrollo relativamente corto.



1.- Introducción a AJAX.



Caso práctico

AJAX es una tecnología crucial en lo que se conoce como web 2.0. A **Antonio** le atrae mucho el tema, ya que ha visto que con **AJAX** se pueden hacer envíos y consultas al servidor, sin tener que recargar las páginas web o cambiar de página, con lo que se consigue que sea todo más interactivo y adaptado a las nuevas tendencias. **Antonio** analiza la tecnología **AJAX**, sus orígenes, y el objeto que se utiliza para realizar las peticiones al servidor y gestionar las respuestas. Su directora **Ada**, le facilita unas direcciones muy interesantes con contenidos y ejemplos de algunas aplicaciones **AJAX** de antiguos proyectos realizados en la empresa.



El término AJAX (JavaScript Asíncrono y XML) es una técnica de desarrollo web, que permite comunicar el navegador del usuario con el servidor, en un segundo plano. De esta forma, se podrían realizar peticiones al servidor sin tener que recargar la página, y podríamos gestionar esas respuestas, que nos permitirían actualizar los contenidos de nuestra página, sin tener que realizar recargas.

El término AJAX se presentó por primera vez en el artículo "A New Approach to Web Applications", publicado por Jesse James Carrett el 18 de febrero de 2005.

AJAX no es una tecnología nueva. Son realmente muchas tecnologías, cada una destacando por su propio mérito, pero que se unen con los siguientes objetivos:

- Conseguir una presentación basada en estándares, usando XHTML, CSS y un uso amplio de técnicas del DOM, para poder mostrar la información de forma dinámica e interactiva.
- Intercambio y manipulación de datos, usando XML y XSLT.
- Recuperación de datos de forma asíncrona, usando el objeto **XMLHttpRequest**.
- Uso de JavaScript, para unir todos los componentes.

Las tecnologías que forman AJAX son:

- XHTML y CSS, para la presentación basada en estándares.
- DOM, para la interacción y manipulación dinámica de la presentación.
- XML, XSLT y JSON, para el intercambio y manipulación de información.
- **XMLHttpRequest**, para el intercambio asíncrono de información.
- JavaScript, para unir todos los componentes anteriores.

El modelo clásico de aplicaciones Web funciona de la siguiente forma: la mayoría de las acciones del usuario se producen en la interfaz, disparando solicitudes HTTP al servidor web. El servidor efectúa un proceso (recopila información, realiza las acciones oportunas), y devuelve una pagina HTML al cliente. Este es un modelo adaptado del uso original de la Web como medio hipertextual, pero a nivel de aplicaciones de software, este tipo de modelo no es necesariamente el más recomendable.

Cada vez que se realiza una petición al servidor, el usuario lo único que puede hacer es esperar, ya que muchas veces la página cambia a otra diferente, y hasta que no reciba todos los datos del servidor, no se mostrará el resultado, con lo que el usuario no podrá interactuar de ninguna manera con el navegador. Con AJAX, lo que se intenta evitar, son esencialmente esas esperas. El cliente podrá hacer solicitudes al servidor, mientras el navegador sigue mostrando la misma página web, y cuando el navegador reciba una respuesta del servidor, la mostrará al cliente y todo ello sin recargar o cambiar de página.

AJAX es utilizado por muchas empresas y productos hoy en día. Por ejemplo, Google utiliza AJAX en aplicaciones como Gmail, Google Suggest, Google Maps., así como Flickr, Amazon, etc.

Son muchas las razones para usar AJAX:

- Está basado en estándares abiertos.
- Su usabilidad.
- Válido en cualquier plataforma y navegador.
- Beneficios que aporta a las aplicaciones web.
- Compatible con Flash.
- Es la base de la web 2.0.
- Es independiente del tipo de tecnología de servidor utilizada.
- Mejora la estética de la web.

1.1.- Requerimientos previos.

A la hora de trabajar con AJAX debemos tener en cuenta una serie de requisitos previos, necesarios para la programación con esta metodología.

Hasta este momento, nuestras aplicaciones de JavaScript no necesitaban de un servidor web para funcionar, salvo en el caso de querer enviar los datos de un formulario y almacenarlos en una base de datos. Es más, todas las aplicaciones de JavaScript que has realizado, las has probado directamente abriéndolas con el navegador o haciendo doble clic sobre el fichero .HTML.

Para la programación con AJAX vamos a necesitar de un servidor web, ya que las peticiones AJAX que hagamos, las haremos a un servidor. Los componentes que necesitamos son:

- Servidor web (apache, ligHTTPd, IIS, etc).
- Servidor de bases de datos (MySQL, Postgresql, etc).
- Lenguaje de servidor (PHP, ASP, etc).

Podríamos instalar cada uno de esos componentes por separado, pero muchas veces lo más cómodo es instalar alguna aplicación que los agrupe a todos sin instalarlos de forma individual. Hay varios tipos de aplicaciones de ese tipo, que se pueden categorizar en dos, diferenciadas por el tipo de sistema operativo sobre el que funcionan:

- servidor LAMP (Linux, Apache, MySQL y PHP).
- servidor WAMP (Windows, Apache, MySQL y PHP).

Una aplicación de este tipo, muy utilizada, puede ser XAMPP (tanto para Windows, como para Linux). Esta aplicación podrás instalarla incluso en una memoria USB y ejecutarla en cualquier ordenador, con lo que tendrás siempre disponible un servidor web, para programar tus aplicaciones AJAX.

Un complemento muy recomendable para la programación con AJAX, es la extensión gratuita **Firebug** de Firefox. Esta extensión es muy útil, ya que con ella podremos detectar errores en las peticiones AJAX, ver los datos que enviamos, en que formato van, qué resultado obtenemos en la petición y un montón de posibilidades más, como inspeccionar todo el DOM de nuestro documento, las hojas de estilo, detectar errores en la programación con JavaScript, etc.

1.2.- Comunicación asíncrona.

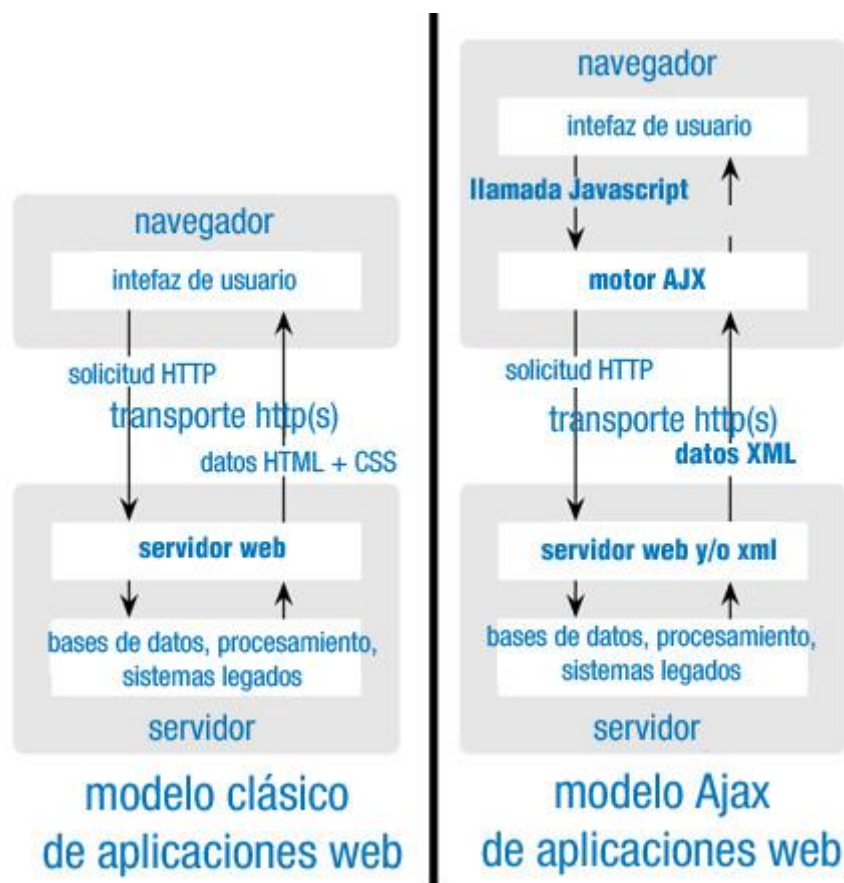
Como ya te comentábamos en la introducción a AJAX, la mayoría de las aplicaciones web funcionan de la siguiente forma:

1. El usuario solicita algo al servidor.
2. El servidor ejecuta los procesos solicitados (búsqueda de información, consulta a una base de datos, lectura de fichero, cálculos numéricos, etc.).
3. Cuando el servidor termina, devuelve los resultados al cliente.

En el paso 2, mientras se ejecutan los procesos en el servidor, el cliente lo único que puede hacer es esperar, ya que el navegador está bloqueado en espera de recibir la información con los resultados del servidor.

Una aplicación AJAX, cambia la metodología de funcionamiento de una aplicación web, en el sentido de que, elimina las esperas y los bloqueos que se producen en el cliente. Es decir, el usuario podrá seguir interactuando con la página web, mientras se realiza la petición al servidor. En el momento de tener una respuesta confirmada del servidor, ésta será mostrada al cliente, o bien se ejecutarán las acciones que el programador de la página web haya definido.

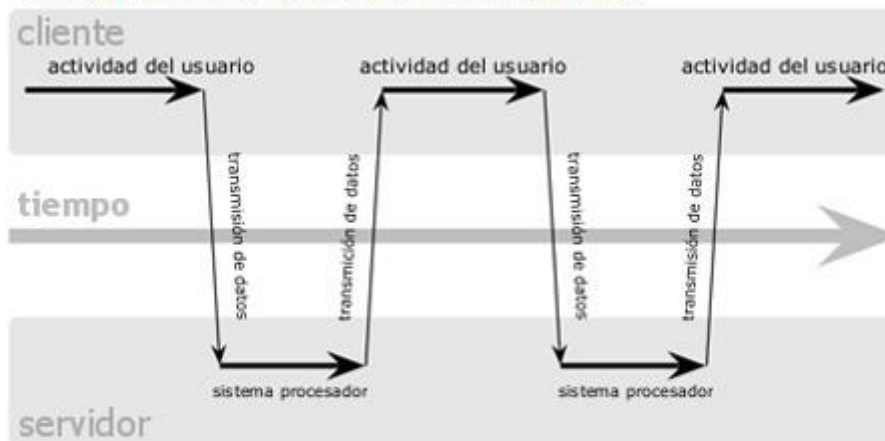
Mira el siguiente gráfico, en el que se comparan los dos modelos de aplicaciones web:



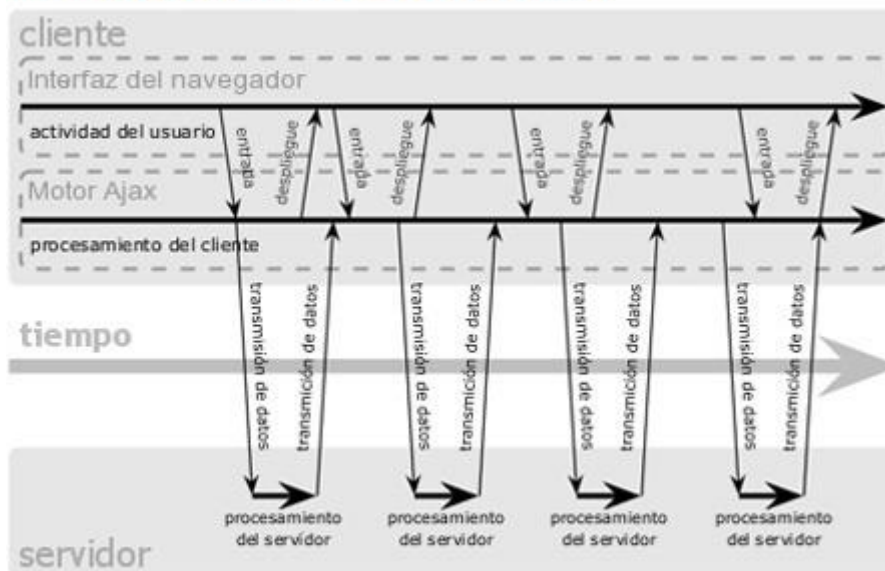
¿Cómo se consigue realizar la petición al servidor sin bloquear el navegador?

Para poder realizar las peticiones al servidor sin que el navegador se quede bloqueado, tendremos que hacer uso del motor AJAX (programado en JavaScript y que antiguamente se encuentra en un frame oculto). Este motor se encarga de gestionar las peticiones AJAX del usuario, y de comunicarse con el servidor. Es justamente este motor, el que permite que la interacción suceda de forma asíncrona (independientemente de la comunicación con el servidor). Así, de esta forma, el usuario no tendrá que estar pendiente del icono de indicador de carga del navegador, o viendo una pantalla en blanco.

modelo clásico de aplicaciones web (síncrono)



modelo Ajax de aplicaciones web (asíncrono)



Cada acción del usuario, que normalmente generaría una petición HTTP al servidor, se va a convertir en una petición AJAX con esa solicitud, y será este motor, el que se encargará de todo el proceso de comunicación y obtención de datos de forma asíncrona con el servidor, y todo ello sin frenar la interacción del usuario con la aplicación.

1.3.- El API XMLHttpRequest.

El corazón de AJAX es una API denominada **XMLHttpRequest** (XHR), disponible en los lenguajes de scripting en el lado del cliente, tales como JavaScript. Se utiliza para realizar peticiones, HTTP o HTTPS, directamente al servidor web, y para cargar las respuestas directamente en la página del cliente. Los datos que recibamos desde el servidor se podrán recibir en forma de texto plano o texto XML. Estos datos, podrán ser utilizados para modificar el DOM del documento actual, sin tener que recargar la página, o también podrán ser evaluados con JavaScript, si son recibidos en formato JSON.

XMLHttpRequest juega un papel muy importante en la técnica AJAX, ya que sin este objeto, no sería posible realizar las peticiones asíncronas al servidor.

El concepto que está detrás del objeto **XMLHttpRequest**, surgió gracias a los desarrolladores de Outlook Web Access (de Microsoft), en su desarrollo de Microsoft Exchange Server 2000. La interfaz

IXMLHttpRequest, se desarrolló e implementó en la segunda versión de la librería MSXML, empleando este concepto. Con el navegador Internet Explorer 5.0 en Marzo de 1999, se permitió el acceso a dicha interfaz a través de ActiveX.

Posteriormente la fundación Mozilla, desarrolló e implementó una interfaz llamada nsXMLHttpRequest, dentro de su motor Gecko. Esa interfaz, se desarrolló adaptándose lo más posible a la interfaz implementada por Microsoft. Mozilla creó un envoltorio para usar esa interfaz, a través de un objeto JavaScript, el cuál denominó **XMLHttpRequest**. El objeto **XMLHttpRequest** fue accesible en la versión 0.6 de Gecko, en diciembre de 2000, pero no fue completamente funcional, hasta Junio de 2002 con la versión 1.0 de Gecko. El objeto **XMLHttpRequest**, se convirtió de hecho en un estándar entre múltiples navegadores, como Safari 1.2, Konqueror, Opera 8.0 e iCab 3.0b352 en el año 2005.

El W3C publicó una especificación-borrador para el objeto **XMLHttpRequest**, el 5 de Abril de 2006. Su objetivo era crear un documento con las especificaciones mínimas de interoperabilidad, basadas en las diferentes implementaciones que había hasta ese momento. La última revisión de este objeto, se realizó en Noviembre de 2009.

Microsoft añadió el objeto **XMLHttpRequest** a su lenguaje de script, con la versión de Internet Explorer 7.0 en Octubre de 2006.

Con la llegada de las librerías cross-browser como jQuery, Prototype, etc, los programadores pueden utilizar toda la funcionalidad de **XMLHttpRequest**, sin codificar directamente sobre la API, con lo que se acelera muchísimo el desarrollo de aplicaciones AJAX.

En febrero de 2008, la W3C publicó otro borrador denominado "**XMLHttpRequest** Nivel 2". Este nivel consiste en extender la funcionalidad del objeto **XMLHttpRequest**, incluyendo, pero no limitando, el soporte para peticiones cross-site, gestión de byte streams, progreso de eventos, etc. Esta última revisión de la especificación, se encuentra en un grupo de trabajo (W3C Working Group) cuya última revisión es del 18 de Noviembre de 2014.

1.3.1.- Creación del objeto XMLHttpRequest.

Para poder programar con AJAX, necesitamos crear un objeto del tipo XMLHttpRequest, que va a ser el que nos permitirá realizar las peticiones en segundo plano al servidor web.

Una vez más, nos vamos a encontrar con el problema de Internet Explorer, que, dependiendo de la versión que utilicemos, tendremos que crear el objeto de una manera o de otra. Aquí tienes un ejemplo de una función cross-browser, que devuelve un objeto del tipo XHR (XMLHttpRequest):

```
////////////////////////////////////
// Función cross-browser para crear objeto XMLHttpRequest
////////////////////////////////////
function objetoXHR()
{
    if (window.XMLHttpRequest)
    {
        // El navegador implementa la interfaz XHR de forma nativa
        return new XMLHttpRequest();
    }
    else if (window.ActiveXObject)
    {
        var versionesIE = new Array('MsXML2.XMLHTTP.5.0', 'MsXML2.XMLHTTP.4.0',
        'MsXML2.XMLHTTP.3.0', 'MsXML2.XMLHTTP', 'Microsoft.XMLHTTP');
    }
}
```

```

for (var i = 0; i < versionesIE.length; i++)
{
    try
    { /* Se intenta crear el objeto en Internet Explorer comenzando
      en la versión más moderna del objeto hasta la primera versión.
      En el momento que se consiga crear el objeto, saldrá del bucle
      devolviendo el nuevo objeto creado. */
      return new ActiveXObject(versionesIE[i]);
    }
    catch (errorControlado) {} //Capturamos el error,
}
}

/* Si llegamos aquí es porque el navegador no posee ninguna forma de crear el objeto.
Emitimos un mensaje de error usando el objeto Error.

Más información sobre gestión de errores en:
HTTP://www.javascriptkit.com/javatutors/trycatch2.shtml */

throw new Error("No se pudo crear el objeto XMLHttpRequest");
}

// para crear un objeto XHR lo podremos hacer con la siguiente llamada.
var objetoAJAX = new objetoXHR();

```

Una opción muy interesante, consiste en hacer una librería llamada, por ejemplo, funciones.js, que contenga el código de tus funciones más interesantes como, `crearEvento()`, `objetoXHR()`, etc. De esta manera, irás creando tus propios recursos, con el código de JavaScript que más uses en tus aplicaciones.

1.3.2.- Métodos del objeto XMLHttpRequest.

El objeto XMLHttpRequest dispone de los siguientes métodos, que nos permitirán realizar peticiones asíncronas al servidor:

Métodos del objeto XMLHttpRequest.	
Metodo	Descripción
abort()	Cancela la solicitud actual.
getAllResponseHeaders()	Devuelve la información completa de la cabecera.
getResponseHeader()	Devuelve la información específica de la cabecera.
open(metodo, url, async, usuario, password)	<p>Especifica el tipo de solicitud, la URL, si la solicitud se debe gestionar de forma asíncrona o no, y otros atributos opcionales de la solicitud.</p> <ul style="list-style-type: none"> • metodo: indicamos el tipo de solicitud: GET o POST. • url: la dirección del fichero al que le enviamos las peticiones en el servidor. • async: true (asíncrona) o false (síncrona). • usuario y password: si fuese necesaria la autenticación en

Métodos del objeto XMLHttpRequest.	
Metodo	Descripción
	el
send (datos)	<ul style="list-style-type: none"> • send(string) Envía la solicitud al servidor. • datos: Se usa en el caso de que estemos utilizando el método POST, como método de envío. Si usamos GET, datos será null.
setRequestHeader()	Añade el par etiqueta/valor a la cabecera de datos que se enviará al servidor.

```
function cargarSync(objeto, url)
{
    if (miXHR)
    {
        alert("Comenzamos la petición AJAX");

        //Si existe el objeto miXHR
        miXHR.open('GET', url, false); //Abrimos la url, false=SINCRONA

        // Hacemos la petición al servidor. Como parámetro del método send:
        // null -> cuando usamos GET.
        // cadena con los datos -> cuando usamos POST
        miXHR.send(null);

        //Escribimos la respuesta recibida de la petición AJAX en el objeto DIV
        textoDIV(objeto, miXHR.responseText);

        alert("Terminó la petición AJAX");
    }
}
```

En esta función se realiza una petición AJAX, pero de forma síncrona (comportamiento normal del navegador). En dicha petición se realiza la carga del fichero indicado en la url (debe ser un fichero perteneciente al mismo DOMinio del servidor). La respuesta (`responseText`), que obtenemos en esa petición, se coloca en un DIV, con la función personalizada `textoDIV` (su código fuente está en el fichero `funciones.js`).

1.3.3.- Propiedades del objeto XMLHttpRequest.

El objeto XMLHttpRequest, dispone de las siguientes propiedades, que nos facilitan información sobre el estado de la petición al servidor, y donde recibiremos los datos de la respuesta devuelta en la petición AJAX:

Propiedades del objeto XMLHttpRequest.	
Propiedad	Descripción
onreadystatechange	Almacena una función (o el nombre de una función), que será llamada automáticamente, cada vez que se produzca un cambio en la propiedad <code>readyState</code> .
readyState	Almacena el estado de la petición XMLHttpRequest. Posibles estados, del 0 al 4: <ul style="list-style-type: none"> • 0: solicitud no inicializada. • 1: conexión establecida con el servidor. • 2: solicitud recibida. • 3: procesando solicitud. • 4: solicitud ya terminada y la respuesta está disponible.
responseText	Contiene los datos de respuesta, como una cadena de texto.
responseXML	Contiene los datos de respuesta, en formato XML.
status	Contiene el estado numérico, devuelto en la petición al servidor (por ejemplo: "404" para "No encontrado" o "200" para "OK").
statusText	Contiene el estado en formato texto, devuelto en la petición al servidor (por ejemplo: "Not Found" o "OK").

```
function cargarAsync(objeto, url)
{
    if (miXHR)
    {
        alert("Comenzamos la petición AJAX");

        //Si existe el objeto miXHR
        miXHR.open('GET', url, true); //Abrimos la url, true=ASINCRONA

        // Hacemos la petición al servidor. Como parámetro:
        // null -> cuando usamos GET.
        // cadena con los datos -> cuando usamos POST
        miXHR.send(null);

        //Escribimos la respuesta recibida de la petición AJAX en el objeto DIV
        textoDIV(objeto, miXHR.responseText);

        alert("Terminó la petición AJAX");
    }
}
```

En esta función se realiza una petición AJAX, pero de forma asíncrona. En dicha petición se realiza la carga del fichero indicado en la url (debe ser un fichero perteneciente al mismo DOMinio del servidor). La respuesta (***responseText***), que obtenemos en esa petición, se coloca en un DIV, con la función personalizada `textoDIV` (su código fuente está en el fichero `funciones.js`). Si ejecutas el ejemplo anterior, verás que no se muestra nada, ya que no hemos gestionado correctamente la respuesta recibida de forma asíncrona. En el siguiente apartado 2, de esta unidad, veremos como corregir ese fallo y realizar correctamente esa operación.

2.- Envío y recepción de datos de forma asíncrona.



Caso práctico

Ahora que **Antonio** ya conoce el objeto **XMLHttpRequest**, con sus propiedades y métodos, se centra en cómo se realiza la petición al servidor de forma asíncrona, y cómo se gestionan los estados y las respuestas que nos devuelve. También estudia qué formatos tiene para enviar datos al servidor, y en qué formatos va a recibir esos resultados.

De entre todos los formatos de recepción de datos, **Juan** recomienda a **Antonio** uno de ellos: el formato JSON. Dicho formato, utiliza la nomenclatura de JavaScript, para enviar los resultados. De esta forma puede utilizar dichos resultados directamente en la aplicación JavaScript, sin tener que realizar prácticamente ningún tipo de conversiones intermedias.



En el apartado 1.3.3., hemos visto un ejemplo de una petición asíncrona al servidor, pero vimos que éramos incapaces de mostrar correctamente el resultado en el contenedor resultados.

```
function cargarAsync(objeto, url)
{
    if (miXHR)
    {
        alert("Comenzamos la petición AJAX");

        //Si existe el objeto miXHR
        miXHR.open('GET', url, true); //Abrimos la url, true=ASINCRONA

        // Hacemos la petición al servidor. Como parámetro:
        // null -> cuando usamos GET.
        // cadena con los datos -> cuando usamos POST
        miXHR.send(null);

        //Escribimos la respuesta recibida de la petición AJAX en el objeto DIV
        textoDIV(objeto, miXHR.responseText);

        alert("Terminó la petición AJAX");
    }
}
```

Si ejecutas el ejemplo del apartado 1.3.3., verás que no se muestra nada en el contenedor resultados, y la razón es por que, cuando accedemos a la propiedad *miXHR.responseText*, ésta no contiene nada. Eso es debido a la solicitud asíncrona. Si recuerdas, cuando hicimos el ejemplo con una solicitud síncrona, se mostró la alerta de "Comenzamos la petición AJAX", aceptaste el mensaje, y justo 2 segundos después recibimos la alerta de "Terminó la petición AJAX". En el modo **síncrono**, el navegador cuando hace la petición al servidor, con el método *miXHR.send()*, se queda esperando hasta que termine la solicitud (y por lo tanto nosotros no podemos hacer otra cosa más que esperar ya que el navegador está bloqueado). Cuando termina la solicitud, pasa a la siguiente línea: **textoDIV(objeto,...)**, y por tanto ya puede mostrar el contenido de **responseText**.

En el modo asíncrono, cuando aceptamos la primera alerta, prácticamente al instante se nos muestra la siguiente alerta. Ésto es así, por que el navegador en una petición asíncrona, no espera a que termine esa solicitud, y continúa ejecutando las siguientes instrucciones. Por eso, cuando se hace la llamada con el método **miXHR.send()**, dentro de la función **cargarAsync()**, el navegador sigue ejecutando las dos instrucciones siguientes, sin esperar a que termine la solicitud al servidor. Y es por

eso que no se muestra nada, ya que la propiedad *responseText*, no contiene ningún resultado todavía, puesto que la petición al servidor está todavía en ejecución. **¿Cuál será entonces la solución?**

Una primera solución que se nos podría ocurrir, sería la de poner un tiempo de espera o retardo, antes de ejecutar la instrucción **textoDIV**. Ésto, lo único que va a hacer, es bloquear todavía más nuestro navegador, y además, tampoco sabemos exactamente lo que va a tardar el servidor web en procesar nuestra solicitud.

La segunda solución, consistiría en detectar cuándo se ha terminado la petición AJAX, y es entonces en ese momento, cuando accederemos a la propiedad *responseText* para coger los resultados. Ésta será la solución, que adoptaremos y que veremos en el apartado 2.1 de esta unidad.

2.1.- Estados de una solicitud asíncrona (parte I).

Cuando se realiza una petición asíncrona, dicha petición va a pasar por diferentes estados (del 0 al 4 - propiedad **readyState** del objeto XHR), independientemente de si el fichero solicitado al servidor, se encuentre o no. Atención: dependiendo del navegador utilizado, habrá algunos estados que no son devueltos.

Cuando dicha petición termina, tendremos que comprobar cómo lo hizo, y para ello evaluamos la propiedad **status** que contiene el estado devuelto por el servidor: 200: *OK*, 404: *Not Found*, etc. Si *status* fue OK ya podremos comprobar, en la propiedad **responseText** o **responseXML** del objeto XHR, los datos devueltos por la petición.

```

////////////////////////////////////
// Función cargarAsync: carga el contenido de la url usando una petición AJAX de forma ASINCRONA.
////////////////////////////////////
function cargarAsync(url)
{
    if (miXHR)
    {
        alert("Comenzamos la petición AJAX");

        //Si existe el objeto  miXHR
        miXHR.open('GET', url, true); //Abrimos la url, true=ASINCRONA

        // En cada cambio de estado(readyState) se llamará a la función estadoPetición
        miXHR.onreadystatechange = estadoPetición;

        // Hacemos la petición al servidor. Como parámetro: null ya que los datos van por GET
        miXHR.send(null);
    }
}

////////////////////////////////////
// Función estadoPetición: será llamada en cada cambio de estado de la petición.
// Cuando el estado de la petición(readyState) ==4 quiere decir que la petición ha terminado.
// Tendremos que comprobar cómo terminó(status): == 200 encontrado, == 404 no encontrado, etc.
// A partir de ese momento podremos acceder al resultado en responseText o responseXML
////////////////////////////////////
function estadoPetición()
{
    switch(this.readyState) // Evaluamos el estado de la petición AJAX

```

```

{ // Vamos mostrando el valor actual de readyState en cada llamada.
  case 0: document.getElementById('estado').innerHTML += "0 - Sin iniciar.<br/>";
  break;
  case 1: document.getElementById('estado').innerHTML += "1 - Cargando.<br/>";
  break;
  case 2: document.getElementById('estado').innerHTML += "2 - Cargado.<br/>";
  break;
  case 3: document.getElementById('estado').innerHTML += "3 - Interactivo.<br/>";
  break;
  case 4: document.getElementById('estado').innerHTML += "4 - Completado.";
    if (this.status == 200) // Si el servidor ha devuelto un OK
    {
      // Escribimos la respuesta recibida de la petición AJAX en el objeto DIV
      textoDIV(document.getElementById("resultados"), this.responseText);
      alert("Terminó la petición AJAX");
    }
    break;
} }

```

2.2.- Estados de una solicitud asíncrona (parte II).

El código del apartado 2.1, fue programado con fines didácticos para mostrar los 4 estados de la petición. El ejemplo completo, con una imagen animada indicadora de la actividad AJAX, se muestra a continuación:

```

////////////////////
// Función cargarAsync: carga el contenido de la url
// usando una petición AJAX de forma ASINCRONA.
////////////////////
function cargarAsync(url)
{
  if (miXHR)
  {
    // Activamos el indicador AJAX.
    document.getElementById("indicador").innerHTML="<img src='AJAX-loader.gif'/>";

    //Si existe el objeto miXHR
    miXHR.open('GET', url, true); //Abrimos la url, true=ASINCRONA

    // En cada cambio de estado(readyState) se llamará a la función estadoPetición
    miXHR.onreadystatechange = estadoPetición;

    // Hacemos la petición al servidor. Como parámetro: null ya que los datos van por GET
    miXHR.send(null);
  }
}

////////////////////
// Función estadoPetición: será llamada en cada cambio de estado de la petición.
// Cuando el estado de la petición(readyState) ==4 quiere decir que la petición ha terminado.
// Tendremos que comprobar cómo terminó(status): == 200 encontrado, == 404 no encontrado, etc.
// A partir de ese momento podremos acceder al resultado en responseText o responseXML

```

```

////////////////////////////////////
function estadoPetición()
{
    // Haremos la comprobación en este orden ya que primero tiene que llegar readyState==4
    // y por último se comprueba el status devuelto por el servidor==200.
    if ( this.readyState==4 && this.status == 200 )
    {
        // Desactivamos el indicador AJAX.
        document.getElementById("indicador").innerHTML="";

        // Metemos en el contenedor resultados las respuestas de la petición AJAX.
        textoDIV(document.getElementById("resultados"), this.responseText);
    }
}
}

```

2.3.- Envío de datos usando método GET.

Vamos a ver un ejemplo, en el que se realiza una petición AJAX a la página procesar.php, pasando dos parámetros: nombre y apellidos, usando el método GET.

```

function iniciar()

{
    // Creamos un objeto XHR.
    miXHR = new objetoXHR();

    // Cargamos de forma asíncrona el texto que nos devuelve la página
    // procesar.php con los parámetros indicados en la URL
    cargarAsync("procesar.php?nombre=Teresa&apellidos=Blanco Ferreiro");
}

////////////////////////////////////
// Función cargarAsync: carga el contenido de la url
// usando una petición AJAX de forma ASINCRONA.
////////////////////////////////////
function cargarAsync(url)
{
    {
        if (miXHR)
        {
            // Activamos el indicador AJAX antes de realizar la petición.
            document.getElementById("indicador").innerHTML="<img src='AJAX-loader.gif'/>";

            //Si existe el objeto miXHR
            miXHR.open('GET', url, true); //Abrimos la url, true=ASINCRONA

            // En cada cambio de estado(readyState) se llamará a la función estadoPetición
            miXHR.onreadystatechange = estadoPetición;

            // Hacemos la petición al servidor. Como parámetro: null ya que los datos van por GET
            miXHR.send(null);
        }
    }
}
////////////////////////////////////

```

```
// Función estadoPetición: será llamada en cada cambio de estado de la petición.
// Cuando el estado de la petición(readyState) ==4 quiere decir que la petición ha terminado.
// Tendremos que comprobar cómo terminó(status): == 200 encontrado, == 404 no encontrado, etc.
// A partir de ese momento podremos acceder al resultado en responseText o responseXML
////////////////////////////////////
function estadoPetición()
{
    // Haremos la comprobación en este orden ya que primero tiene que llegar readyState==4
    // y por último se comprueba el status devuelto por el servidor==200.
    if ( this.readyState==4 && this.status == 200 )
    {
        // Desactivamos el indicador AJAX.
        document.getElementById("indicador").innerHTML="";

        // Metemos en el contenedor resultados las respuestas de la petición AJAX.
        textoDIV(document.getElementById("resultados"), this.responseText);
    }
}
```

En la petición **GET**, los parámetros que pasemos en la solicitud, se enviarán formando parte de la URL. Por ejemplo: *procesar.php?nombre=Teresa&apellidos=Blanco Ferreiro*. Cuando realizamos la petición por el método GET, te recordamos una vez más, que pondremos *null* como parámetro del método *send*, ya que los datos son enviados a la página *procesar.php*, formando parte de la URL: **send(null)**.

2.4.- Envío de datos usando método POST.

Vamos a ver un ejemplo en el que se realiza una petición AJAX, a la página *procesar.php* pasando dos parámetros: nombre y apellidos, usando el método **POST**.

```
function iniciar()
{
    // Creamos un objeto XHR.
    miXHR = new objetoXHR();

    // Cargamos de forma asíncrona el texto que nos devuelve la página
    // procesar.php
    // En este caso sólo ponemos los parámetros que pasaremos a la página procesar.php
    cargarAsync("nombre=Teresa&apellidos=Blanco Ferreiro");
}

////////////////////////////////////
// Función cargarASync: carga el contenido con los parametros
// que se le vana a pasar a la petición AJAX de forma ASINCRONA.
////////////////////////////////////
function cargarAsync(parametros)
{
    if (miXHR)
    {
        // Activamos el indicador AJAX antes de realizar la petición.
        document.getElementById("indicador").innerHTML="<img src='AJAX-loader.gif'/>";
    }
}
```

```
// Abrimos la conexión al servidor usando método POST y a la página procesar.php
miXHR.open('POST', "procesar.php", true); // Abrimos la url, true=ASINCRONA

// En cada cambio de estado(readyState) se llamará a la función estadoPetición
miXHR.onreadystatechange = estadoPetición;

// En las peticiones POST tenemos que enviar en la cabecera el Content-Type
//ya que los datos se envían formando parte de la cabecera.
miXHR.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");

// Hacemos la petición al servidor con los parámetros: nombre=Teresa&apellidos=Blanco...
miXHR.send(parametros);
}
}
```

En este ejemplo, tuvimos que realizar los siguientes cambios para adaptarlo al método POST:

- La función **cargarAsync()**, recibirá los parámetros por **POST** en lugar de **GET**.
- El método `.open` se modifica por:

```
miXHR.open('POST', "procesar.php", true);
```

- Tenemos que crear una cabecera con el tipo de contenido que vamos a enviar, justo antes de enviar la petición con el método **.send()**:

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

- En el método `.send()` escribiremos los parámetros (**nombre=Teresa&apellidos=Blanco Ferreiro**) que serán enviados por el método POST:

```
miXHR.send(parametros).
```

2.5.- Recepción de datos en formato XML.

Cuando realizamos una petición AJAX, que nos devuelve las respuestas en formato XML, dichos datos los tendremos que consultar en la propiedad **responseXML** del objeto XHR.

En la función **iniciar()**, le hemos dicho que cargue de forma asíncrona, empleando el método **GET**, el fichero `datosXML.php`. Esta aplicación PHP, nos devolverá un fichero XML, con una lista de Cd de música con el artista, país, compañía, etc.

Instrucción de carga del fichero `datosXML.php`: **cargarAsync("datosXML.php");**

Si queremos cargar directamente un fichero XML, y conocemos su nombre, podremos escribir directamente:

```
cargarAsync("catalogo.XML");
```

En la función de `estadoPetición()`, cuando `readyState` es 4 y el `status` es **OK** (200), accedemos a los resultados de la petición AJAX, en la propiedad `responseXML`. Para gestionar los datos XML, tendremos que recorrerlos empleando los métodos del DOM, ya que un fichero XML comparte la estructura en

árbol de un documento HTML, y podemos utilizar, por tanto, los mismos métodos que empleamos para recorrer el DOM HTML.

En nuestro caso, lo primero que vamos a hacer es recorrer los elementos , que son los que contienen toda la información referente a los cd's de música:

```
// Almacenamos el fichero XML en la variable resultados.
resultados=this.responseXML;

// Tenemos que recorrer el fichero XML empleando los métodos del DOM
// Array que contiene todos los CD's del fichero XML
CDs= resultados.documentElement.getElementsByTagName("CD");
```

Haremos un bucle para recorrer todos los cd's del catálogo, y dentro de cada uno, imprimiremos los datos que nos interesen:

```
/ Hacemos un bucle para recorrer todos los elementos CD.
for (i=0;i<CDs.length;i++)
{
.....
```

Dentro de cada CD, accederemos al elemento que nos interese e imprimiremos su contenido. Para imprimir el contenido de cada nodo, tendremos que hacerlo con el comando `try { } catch {}`, ya que si intentamos acceder a un nodo que no tenga contenido, nos dará un error de JavaScript, puesto que el elemento hijo no existe, y entonces se detendrá la ejecución de JavaScript y no imprimirá nuestro listado.

```
// Para cada CD leemos el título
titulos=CDs[i].getElementsByTagName("TITLE");

try    // Intentamos acceder al contenido de ese elemento
{
    salida+=" " + titulos[0].firstChild.nodeValue + "</td>"; } catch (er)    // En el caso de que no tenga contenido ese elemento imprimimos un espacio en blanco. {     salida+= "<td>&nbsp;</td>"; } |
```

2.6.- Recepción de datos en formato JSON (parte I).

Otro formato de intercambio muy utilizado en AJAX, es el formato JSON. JSON es un formato de intercambio de datos, alternativo a XML, mucho mas simple de leer, escribir e interpretar. Significa **Javascript Object Notation**, y consiste en escribir los datos en formato de Javascript. Vamos a hacer un poco de repaso:

Arrays

Se pueden crear con corchetes:

```
var Beatles = ["Paul","John","George","Ringo"];
```

Con new Array():

```
var Beatles = new Array("Paul","John","George","Ringo");
```

O también de la siguiente forma:

```
var Beatles = { "Paul","John","George","Ringo"};
```

Objetos

Un objeto literal se puede crear entre llaves: { propiedad1:valor, propiedad2:valor, propiedad3:valor }

```
var Beatles = {  
  "Country" : "England",  
  "YearFormed" : 1959,  
  "Style" : "Rock'n'Roll"  
}
```

Que será equivalente a:

```
var Beatles = new Object();  
  
Beatles.Country = "England";  
Beatles.YearFormed = 1959;  
Beatles.Style = "Rock'n'Roll";
```

Y para acceder a sus propiedades lo podemos hacer con:

```
alert(Beatles.Style); // Notación de puntos  
alert(Beatles["Style"]); // Notación de corchetes
```

Los objetos también pueden contener arrays literales: [...]

```
var Beatles = {  
  "Country" : "England",  
  "YearFormed" : 1959,  
  "Style" : "Rock'n'Roll",  
  "Members" : ["Paul","John","George","Ringo"]  
}
```

Y los arrays literales podrán contener objetos literales a su vez: {...} , {...}

```
var Rockbands =  
[  
  { "Name" : "Beatles", "Country" : "England", "YearFormed" : 1959, "Style" : "Rock'n'Roll",  
    "Members" :  
    ["Paul","John","George","Ringo"] } , { "Name" : "Rolling Stones", "Country" : "England",  
    "YearFormed" : 1962, "Style" : "Rock'n'Roll", "Members" : ["Mick","Keith","Charlie","Bill"] }  
]
```

La sintaxis de JSON es como la sintaxis literal de un objeto, excepto que, esos objetos no pueden ser asignados a una variable. JSON representará los datos en sí mismos. Por lo tanto el objeto *Beatles* que vimos antes se definiría de la siguiente forma:

```
{
  "Name" : "Beatles",
  "Country" : "England",
  "YearFormed" : 1959,
  "Style" : "Rock'n'Roll",
  "Members" : ["Paul","John","George","Ringo"]
}
```

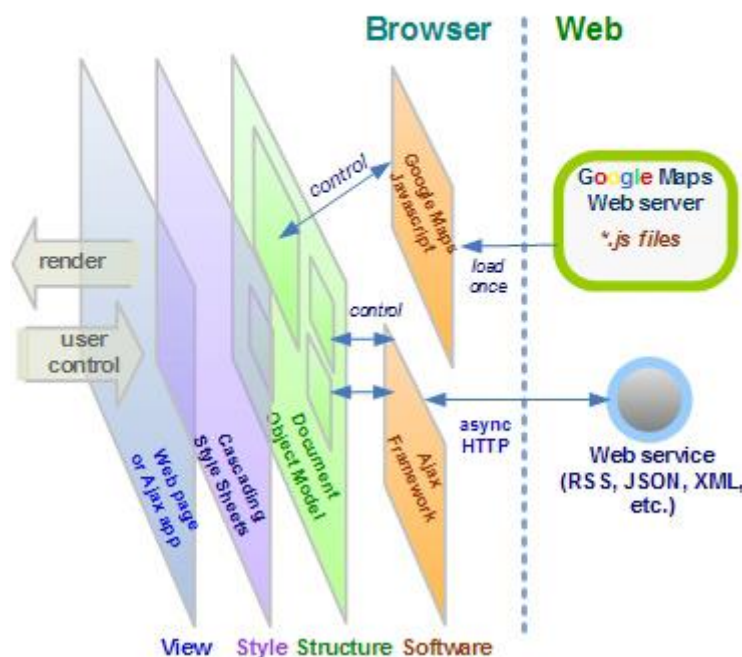
Una cadena JSON es, simplemente, una cadena de texto, y no un objeto en sí misma. Necesita ser convertida a un objeto antes de poder ser utilizada en JavaScript. Ésto se puede hacer con la función `eval()` de JavaScript y también se pueden usar lo que se conoce como analizadores JSON, que facilitarán esa conversión.

2.7.- Recepción de datos en formato JSON (parte II).

Veamos un ejemplo completo en el que recibimos, con AJAX, los datos procedentes de un listado de una tabla MySQL en formato JSON.

Si quieres probar el ejemplo, necesitas un servidor web, PHP y MySQL. Puedes instalarte por ejemplo XAMPP en cualquiera de sus versiones para Windows o Linux.

Anatomy of a Google Maps Mashup



En este ejemplo, se realiza una consulta a una tabla (se adjunta código SQL de creación de la base de datos, usuario, contraseña y la tabla con los datos). La página PHP, devolverá los resultados en una cadena de texto, que contiene un array de objetos literales en formato JSON.

Código de PHP del fichero *datosJSON.php*:

```
// Consulta SQL para obtener los datos de los centros.
$sql="select * from centros order by nombrecentro";
$resultados=mysql_query($sql,$conexion) or die(mysql_error());

while ( $fila = mysql_fetch_array($resultados, MYSQL_ASSOC))
{
    // Almacenamos en un array  cada una de las filas que vamos leyendo del recordset.
    $datos[]=$fila;
}

echo JSON_encode($datos); // función de PHP que convierte a formato JSON el array.
....
```

Nuestra aplicación de JavaScript recibe, por AJAX, esos datos, en la propiedad *responseText*. Para poder utilizar directamente esos datos en JavaScript, lo que tenemos que hacer es evaluar la expresión (cadena de texto, que recibimos de la página *datosJSON.php*). La expresión JSON contenía un array de objetos literales, por lo que tenemos que crear una variable, para asignar ese array y poder recorrerlo.

Para evaluar la expresión lo hacemos con la función **eval()** de JavaScript:

```
// Asignamos a la variable resultados la evaluación de.responseText
var resultados=eval( '(' +this.responseText+')');

// Hacemos un bucle para recorrer todos los objetos literales recibidos en el array resultados y mostrar su contenido.
for (var i=0; i < resultados.length; i++)
{
    objeto = resultados[i];
    texto+="<tr><td>" +objeto.nombrecentro+"</td><td>"
+objeto.localidad+"</td><td>"
+objeto.provincia+"</td><td>" +objeto.telefono+"</td><td>" +objeto.fechavisita + "</td><td>" +
objeto.numvisitantes+ "</td></tr>";
}
```

3.- Librerías cross-browser para programación AJAX.



Caso práctico

El estudio del objeto para trabajar con AJAX, está comenzando a dar sus frutos. Lo que más le fastidia a **Antonio**, es que necesita programar bastante código, y aunque puede crear alguna librería para acelerar la programación, también ve que las diferentes incompatibilidades, entre navegadores, no van a ayudar nada en esta labor. Por esta razón está un poco desilusionado, por que le va a suponer bastante trabajo, aunque los resultados merecen la pena.

En ese momento llega **Juan**, y le da la sorpresa que le había comentado hace unos días. Le facilita un pequeño tutorial sobre la librería jQuery, que le va a permitir hacer peticiones AJAX utilizando prácticamente una línea de código. No tendrá que preocuparse por temas de cross-browsing y, además, la misma librería le facilitará métodos para hacer todo tipo de efectos, animaciones, etc. Esta librería cuenta además con infinidad de complementos gratuitos, que permiten hacer prácticamente cualquier cosa en muy poco tiempo.



La programación con AJAX, es uno de los pilares de lo que se conoce como web 2.0, término que incluye a las aplicaciones web que facilitan el compartir información, la interoperabilidad, el diseño centrado en el usuario y la colaboración web. Ejemplos de la web 2.0, pueden ser las comunidades web, los servicios web, aplicaciones web, redes sociales, servicios de alojamiento de vídeos, wikis, blogs, mashup, etc.

Gracias a las aplicaciones web 2.0, se han desarrollado gran cantidad de utilidades/herramientas/frameworks para el desarrollo web con JavaScript, DHTML (HTML dinámico) y AJAX. La gran ventaja de usar alguna librería o framework para AJAX, es la del ahorro de tiempo y código, en nuestras aplicaciones. Veremos que con algunas librerías vamos a realizar peticiones AJAX, con una simple instrucción de código sin tener que preocuparnos de crear el objeto XHR, ni gestionar el código de respuesta del servidor, los estados de la solicitud, etc.

Otra de las ventajas que nos aportan este tipo de librerías, es la de la compatibilidad entre navegadores (cross-browser). De esta forma tenemos un problema menos, ya que la propia librería será capaz de crear la petición AJAX de una forma u otra, dependiendo del navegador que estemos utilizando.

A principios del año 2008 Google liberó su API de librerías AJAX, como una red de distribución de contenido y arquitectura de carga, para algunos de los frameworks más populares. Mediante esta API se eliminan las dificultades a la hora de desarrollar mashups en JavaScript. Se elimina el problema de alojar las librerías (ya que están centralizadas en Google), configurar las cabeceras de cache, etc. Esta API ofrece acceso a las siguientes librerías Open Source, realizadas con JavaScript:

- jQuery. Fácil y muy usada.
- Angular. Tenía mucho futuro y estaba superando en uso en proyectos a jQuery. Pero Google va a utilizar TypeScript, un JavaScript con tipos creado por Microsoft, y van a romper compatibilidad. Así que Angular 2.0 será un proceso de volver aprender, al menos eso dice Google en el momento de la creación de este documento.
- Backbone.js. No está nada mal.
- scriptaculous.
- mooTools.
- Dojo. Siempre muy prometedora.
- prototype.js (pero ya lleva 11 meses sin actualizaciones).
- Fuera del objetivo del módulo podéis programar aplicaciones ligeras para dispositivos móviles con Cordova (PhoneGap libre) (Android, ios, Windows Phone, Windows 8, etc..).

Los scripts de estas librerías están accesibles directamente utilizando la URL de descarga, o a través del método `google.load()` del cargador de la API AJAX de Google.

Hay muchísimas librerías que se pueden utilizar para programar AJAX, dependiendo del framework que utilicemos. En el siguiente listado se ven las que crecieron mucho en el pasado 2014.

<http://webdesignmoo.com/2014/21-best-javascript-frameworks-2014/>

Nosotros nos vamos a centrar en el uso de la librería jQuery, por ser una de las más utilizadas hoy en día por empresas como Google, DELL, digg, NBC, CBS, NETFLIX, mozilla.org, wordpress, drupal, etc.

3.1.- Introducción a jQuery (parte I).

Fotografía de una persona al lado de un cartel que anuncia una conferencia de jQuery. jQuery es un framework JavaScript, que nos va a simplificar muchísimo la programación. Como bien sabes, cuando usamos JavaScript tenemos que preocuparnos de hacer scripts compatibles con varios navegadores y, para conseguirlo, tenemos que programar código compatible.

jQuery nos puede ayudar muchísimo a solucionar todos esos problemas, ya que nos ofrece la infraestructura necesaria para crear aplicaciones complejas en el lado del cliente. Basado en la filosofía de "*escribe menos y produce más*", entre las ayudas facilitadas por este framework están: la creación de interfaces de usuario, uso de efectos dinámicos, AJAX, acceso al DOM, eventos, etc. Además esta librería cuenta con infinidad de plugins, que nos permitirán hacer presentaciones con imágenes, validaciones de formularios, menús dinámicos, drag-and-drop, etc.

Esta librería es gratuita, y dispone de licencia para ser utilizada en cualquier tipo de plataforma, personal o comercial. El fichero tiene un tamaño aproximado de 31 KB, y su carga es realmente rápida. Además, una vez cargada la librería, quedará almacenada en caché del navegador, con lo que el resto de páginas que hagan uso de la librería, no necesitarán cargarla de nuevo desde el servidor.

[Página Oficial de descarga de la librería jQuery.](#)

Para saber más : [Documentación oficial de la librería jQuery.](#)

Para poder programar con jQuery, lo primero que tenemos que hacer es cargar la librería. Para ello, podemos hacerlo de dos formas:

Cargando la librería directamente desde la propia web de jQuery con la siguiente instrucción:

```
<script type="text/javascript" src="HTTP://code.jquery.com/jquery-latest.js"></script>
```

De esta forma, siempre nos estaremos descargando la versión más actualizada de la librería. El único inconveniente, es que necesitamos estar conectados a Internet para que la librería pueda descargarse.

Cargando la librería desde nuestro propio servidor:

```
<script type="text/javascript" src="jquery.js"></script>
```

De esta forma, el fichero de la librería estará almacenado como un fichero más de nuestra aplicación, por lo que no necesitaremos tener conexión a Internet (si trabajamos localmente), para poder usar la librería. Para poder usar este método, necesitaremos descargarnos el fichero de la librería desde la página de jQuery (jquery.com). Disponemos de dos versiones de descarga: la *versión de producción* (comprimida para ocupar menos tamaño), y la *versión de desarrollo* (descomprimida). Generalmente descargaremos la versión de producción, ya que es la que menos tamaño ocupa. La versión de desarrollo tiene como única ventaja que nos permitirá leer, con más claridad, el código fuente de la librería (si es que estamos interesados en modificar algo de la misma).

La clave principal para el uso de jQuery radica en el uso de la función **\$()**, que es un alias de **jQuery()**. Esta función se podría comparar con el clásico *document.getElementById()*, pero con una diferencia muy importante, ya que soporta selectores CSS, y puede devolver arrays. Por lo tanto **\$()** es una versión mejorada de *document.getElementById()*.

Esta función **\$("selector")**, acepta como parámetro una cadena de texto, que será un selector CSS, pero también puede aceptar un segundo parámetro, que será el contexto en el cuál se va a hacer la búsqueda del selector citado. Otro uso de la función, puede ser el de **\$(function){..}**; equivalente a la instrucción **\$(document).ready (function() {...})**; que nos permitirá detectar cuando el DOM está completamente cargado.

Verás un ejemplo de como usar estas instrucciones en apartado siguiente 3.2.

3.2.- Introducción a jQuery (parte II).

Vamos a ver en este apartado, un ejemplo programado por el método tradicional, y su equivalencia, usando la librería jQuery:

Ejemplo usando el método tradicional con JavaScript:

```

... Aquí irán las cabeceras y clase .colorido

<script type="text/javascript" src="funciones.js"></script>
<script type="text/javascript">
////////////////////////////////////
// Cuando el documento esté cargado completamente llamamos a la función iniciar().
////////////////////////////////////
crearEvento(window,"load",iniciar);
////////////////////////////////////

function iniciar()
{
    var tabla=document.getElementById("mitabla"); // Seleccionamos la tabla.
    var filas= tabla.getElementsByTagName("tr"); // Seleccionamos las filas de la tabla.

    for (var i=0; i<filas.length; i++)
    {
        if (i%2==1)    // Es una fila impar
        {
            // Aplicamos la clase .colorido a esas filas.
            filas[i].setAttribute('class','colorido');
        }
    }
}
</script>
</head>
<body>
    .. Aquí irá la tabla ...
</body>
</HTML>

```

Ejemplo equivalente al anterior, programado usando la librería jQuery:

```

... Aquí irán las cabeceras y clase .colorido

```

```
<script type="text/javascript" src="HTTP://code.jquery.com/jquery-latest.js"></script>
<script type="text/javascript">
// También podríamos poner $(function) {...});
$(document).ready(function() // Cuando el documento esté preparado se ejecuta esta función.
{
    // Seleccionamos las filas impares contenidas dentro de mitabla y le aplicamos la clase colorido.
    $("#mitabla tr:nth-child(even)").addClass("colorido");
});
</script>
</head>
<body>
    .. Aquí irá la tabla ...
</body>
</HTML>
```

Como puedes observar, la reducción de código es considerable. Con 2 instrucciones, hemos conseguido lo mismo, que hicimos en el ejemplo anterior con 7 (sin contar el código de crearEvento del fichero funciones.js).

3.3.- Introducción a jQuery (parte III).

En la unidad anterior se aprendió como gestiona JavaScript y w3c los eventos y vimos una serie de problemas con los eventos y la compatibilidad entre navegadores. jQuery surgió entre otras cosas para ayudarnos a evitar estos problemas.

Veamos por ejemplo como se haría una asignación de una función a un evento click en un botón:

```
<!doctype html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Básico de JQuery</title>
    <script type="text/javascript" src="HTTP://code.jquery.com/jquery-latest.js"></script>
</head>

<body>
    <h1> Botón Fácil</h1>
    <button>Mi botón</button>
    <script lang="text/javascript">
        $("button").click(function() {
            $("body").append("<br />Se ha pulsado el botón");
        });
    </script>
</body>
</html>
```

Si os fijáis en el método click se le pasa un parámetro que es una función anónima. Es algo común en muchos códigos, pero no recomendable en muchas circunstancias. Es mejor que creéis una función y la llamamos de esta forma:


```
<script lang="text/javascript">
  function pulsado() {
    $("body").append("<br />Se ha pulsado el botón");
  };
  $("button").click(pulsado);
</script>
```

Para una gran mayoría de eventos w3c, jQuery dispone de su método para el evento. Por ejemplo como `mouseenter`, `mouseleave`, `resize`, `focus`, `scroll` y un largo etcétera. Se pueden consultar en uno de los enlaces que se encuentra más adelante.

Sin embargo hay uno que es muy interesante denominado `ready`. Este evento equivalente al `[on]load` que hemos estado utilizando hasta ahora en el curso para asegurarnos que el DOM está completamente cargado. Por ejemplo, si introducimos el anterior código en el `head` (o en otro archivo externo) para asignar el evento `click` a la función `pulsado()` hay que realizar la asignación del evento dentro de un `ready` tal y como veremos en el siguiente ejemplo.

```
<!doctype html>
<html>

<head>
  <meta charset="UTF-8">
  <title>Básico de JQuery</title>
  <script type="text/javascript" src="HTTP://code.jquery.com/jquery-latest.js"></script>
  <script lang="text/javascript">
    function pulsado() {
      $("body").append("<br />Se ha pulsado el botón");
    };
    // La llamada al siguiente método es equivalente al onload y al load
    $(document).ready(function() {
      // Aquí dentro se ejecuta lo que se quiera que se haga
      // al cargar el DOM.
      $("button").click(pulsado);
    });
  </script>
</head>

<body>
  <h1>Alguna cosas </h1>
  <button>Mi botón</button>
</body>
</html>
```

La llamada a `ready` se puede substituir por la siguiente:

```
$(function() { // Sustituye al ready()
  $("button").click(pulsado);
});
```

3.4.- Introducción a jQuery (parte IV).

Para terminar esta introducción vamos a ver como se gestionan los eventos de una forma más moderna con el uso del método on.

El método on te permite trabajar con los eventos w3c de forma similar a lo que se hace con addEventListener. Veamos el ejemplo anterior actualizado:

```
<script lang="text/javascript">
  function pulsado() {
    $("body").append("<br />Se ha pulsado el botón");
  };
  $(function() {
    $("button").on("click",pulsado);
  });
</script>
```

Es una forma más elegante y creo que comprensible que en los métodos visto en el apartado 3.3

Otra ventaja es la posibilidad de añadir distintos eventos y elementos para ser procesados por la misma función. Estos argumentos deben estar dentro de un array literal tal y como se ve en el siguiente ejemplo:

```
<!doctype html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Gestión de Eventos con JQuery</title>
  <script type="text/javascript" src="HTTP://code.jquery.com/jquery-latest.js"> </script>
  <script>
    $(function() {
      console.log("hace algo");
      $("#nombre,#apellidos").on({ //on ha sustituido a bind a partir del jquery 1.7
        "focus": miGestorEventos,
        "blur": miGestorEventos
      });
    });
    function miGestorEventos(evento) {
      console.log(evento);
      if (evento.type == "focus") { // Se se ha producido el evento focus
        $(this).css("backgroundColor", "blue");
      } else { // Si se ha producido el evento blur
        $(this).css("backgroundColor", "white");
      }
    }
  </script>
</head>
```

```
<body>
  <form>
    <label for="nombre">Nombre</label>
    <input type="text" id="nombre" name="entrada">
    <br />
    <label for="apellidos">Apellidos</label>
    <input type="text" id="apellidos" name="entrada">
  </form>
<br />
</body>
</html>
```

Con esto sólo se ha visto un poco de jQuery. Hay mucho más que aprender ya que es una librería excepcionalmente útil. Os recomiendo encarecidamente que os leáis los enlaces del siguiente para que jQuery os facilite la vida tanto como a mí.

3.5.- Función \$.ajax() en jQuery.

La principal función para realizar peticiones AJAX en jQuery es \$.AJAX() (importante no olvidar el punto entre \$ y AJAX()). Ésta es una función de bajo nivel, lo que quiere decir que disponemos de la posibilidad de configurar, prácticamente todos los parámetros de la petición AJAX, y será, por tanto, equivalente a los métodos clásicos que usamos en la programación tradicional.

La sintaxis de uso es: \$.AJAX(opciones)

En principio, esta instrucción parece muy simple, pero el número de opciones disponibles, es relativamente extenso. Ésta es la estructura básica:

```
$.AJAX({
  url: [URL],
  type: [GET/POST],
  success: [function callback éxito(data)],
  error: [function callback error],
  complete: [function callback error],
  ifModified: [bool comprobar E-Tag],
  data: [mapa datos GET/POST],
  async: [bool que indica sincronía/asincronía]
});
```

Por ejemplo:

```
$.AJAX({
  url: '/ruta/pagina.php',
  type: 'POST',
  async: true,
  data: 'parametro1=valor1&parametro2=valor2',
  success: function (respuesta)
  {
    alert(respuesta);
  }
});
```

```

    },
    error: mostrarError
  });

```

Veamos algunas propiedades de la función \$.AJAX() de jQuery:

Propiedades de la función \$.ajax() de jQuery.		
Nombre	Tipo	Descripción
url	String	La URL a la que se le hace la petición AJAX.
type	String	El método HTTP a utilizar para enviar los datos: POST o GET. Si se omite se usa GET por defecto.
data	Object	Un objeto en el que se especifican parámetros que se enviarán en la solicitud. Si es de tipo GET, los parámetros irán en la URL. Si es POST, los datos se enviarán en las cabeceras. Es muy útil usar la función serialize(), para construir la cadena de datos.
dataType	String	Indica el tipo de datos que se espera que se devuelvan en la respuesta: XML, HTML, JSON, JSONp, script, text (valor por defecto en el caso de omitir dataType).
success	Function	Función que será llamada, si la respuesta a la solicitud terminó con éxito.
error	Function	Función que será llamada, si la respuesta a la solicitud devolvió algún tipo de error.
complete	Function	Función que será llamada, cuando la solicitud fue completada.

Por último veamos un ejemplo en el que se utilice JSON. Para acceder a un archivo (en este caso ya está creado en el servidor), de películas de cine con para que podáis ver como funciona:

Contenido de ajaxpelis.json

```

[
  { "titulo": "Terminator", "genero": "Ciencia Ficción" },
  { "titulo": "Alien", "genero": "Terror" },
  { "titulo": "La Salchica Peleona", "genero": "comedia" }
]

```

Archivo html:

```

<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>Jquery método ajax con JSON.</title>
  <script src="jquery-1.11.1.js"></script>

  <script>
    $(function () {
      $("button").click(function () { // Se se pulsa el boton
        var cadena = "";
        $("#cambia").html("<img src='ajax-loader.gif'>"); // Mientras se carga
        // El método ajax recibe un objeto literal
        $.ajax({

```

```

        url: "ajaxpelis.json",
        // data: { edadrecomendada: 16}, // El post al servidor No hace falta para este ejemplo
        type: "GET",
        dataType: "json",
        // Si se produce correctamente
        success: function (datos_devueltos) {
            $("#cambia").html("<br />");
            for (var i = 0; i < datos_devueltos.length; i++) {
                $("#cambia").append("Título:" + datos_devueltos[i].titulo + "<br />" + "Genero:"
+ datos_devueltos[i].genero + "<br />");
            }
        },
        // Si la petición falla
        error: function (xhr, estado, error_producido) {
            console.log("Error producido: " + error_producido);
            console.log("Estado: " + estado);

        },
        //Tanto si falla como si funciona como sino funciona.
        complete: function (xhr, estado) {
            console.log("Petición completa");
        }
    });

});
});
</script>
</head>
<body>
    <button type="button">jQuery Ajax JSON.</button>
    <div id="cambia">
        <h2>Cambia el contenido</h2>
    </div>
</body>
</html>

```

Si no os funciona, asegurados que estáis el archivo html se llama desde el mismo dominio de el que se llama el html, sino os va a dar errores de permisos. Y buscar una imagen AJAX de carga gif para que los usuarios sepan que se está procesando una petición AJAX.

3.6.- El método .load() y las funciones \$.post() , \$.get() y \$.getJSON() en jQuery.

La función **\$.AJAX()** es una función muy completa, y resulta bastante pesada de usar. Su uso es recomendable, para casos muy concretos, en los que tengamos que llevar un control exhaustivo de la petición AJAX. Para facilitarnos el trabajo, se crearon 3 funciones adicionales de alto nivel, que permiten realizar peticiones y gestionar las respuestas obtenidas del servidor:

El método .load()

Este método, es la forma más sencilla de obtener datos desde el servidor, ya que de forma predeterminada, los datos obtenidos son cargados en el objeto al cuál le estamos aplicando el método.

Su sintaxis es: **.load(url, [datos], [callback])**

La función **callback** es opcional, y es ahí donde pondremos la función de retorno, que será llamada una vez terminada la petición. En esa función realizaremos tareas adicionales, ya que la acción por defecto de cargar en un objeto el contenido devuelto en la petición, la realiza el propio método *load()*.

Ejemplos:

```
$("#noticias").load("feeds.HTML");  
// carga en el contenedor con id noticias lo que devuelve la página feeds.HTML.  
  
$("#objectID").load("test.php", { 'personas[]': ["Juan", "Susana"] } );  
// Pasa un array de datos al servidor con el nombre de dos personas.
```

Cuando se envían datos en este método, se usará el método POST. Si no se envían datos en la petición, se usará el método GET.

Debes conocer: [Más información sobre el método .load\(\) en jQuery.](#)

La función \$.post()

Nos permite realizar peticiones AJAX al servidor, empleando el método POST. Su sintaxis es la siguiente:

\$.post(url, [datos], [callback], [tipo])

Ejemplos:

```
$.post("test.php");  
  
$.post("test.php", { nombre: "Juana", hora: "11am" } );  
  
$.post("test.php", function(resultados) {  
    alert("Datos Cargados: " + resultados);  
});
```

Debes conocer: [Más información sobre el método .post\(\) en jQuery.](#)

La función \$.get() y \$.getJSON()

Hacen prácticamente lo mismo que POST, y tienen los mismos parámetros, pero usan el método GET para enviar los datos al servidor. Si recibimos los datos en formato JSON, podemos emplear *\$.getJSON()* en su lugar.

\$.get(url, [datos], [callback], [tipo]) | \$.getJSON(url, [datos], [callback], [tipo])

Debes conocer: [Más información sobre el método .get\(\) en jQuery.](#)

3.7.- Herramientas adicionales en programación AJAX.

Cuando programamos en AJAX, uno de los inconvenientes que nos solemos encontrar, es el de la detección de errores. Estos errores pueden venir provocados por fallos de programación en JavaScript, fallos en la aplicación que se ejecuta en el servidor, etc.

Para poder detectar estos errores, necesitamos herramientas que nos ayuden a encontrarlos. En la programación con JavaScript, los errores los podemos detectar con el propio navegador. Por ejemplo, en el navegador Firefox para abrir la consola de errores, lo podemos hacer desde el menú Herramientas, o bien pulsando las teclas **CTRL+Mayúsc.+J** (en Windows) o F12 si se dispone de una versión de firebug moderna. En Chrome también se utiliza con F12. En la consola, se nos mostrarán todos los errores que se ha encontrado durante la ejecución de la aplicación. En Internet Explorer versión 9 y posteriores, podemos abrir la **Herramienta de Desarrollo**, pulsando la tecla F12. Desde esta herramienta se pueden consultar los errores de JavaScript, activar los diferentes modos de compatibilidad entre versiones de este navegador, deshabilitar CSS, JavaScript, etc.

Para la detección de errores en AJAX, necesitamos herramientas adicionales o complementos. Para Firefox disponemos de un complemento denominado Firebug. Este complemento nos va a permitir hacer infinidad de cosas: detectar errores de JavaScript, depurar código, analizar todo el DOM del documento en detalle, ver y modificar el código CSS, analizar la velocidad de carga de las páginas, etc. Además, también incorpora en su consola, la posibilidad de ver las peticiones AJAX que se están realizando al servidor: se pueden ver los datos enviados, su formato, los datos recibidos, los errores, etc. Si por ejemplo se produce algún tipo de error en la petición al servidor, en la consola podremos verlo y así poder solucionar ese fallo.

3.8.- Plugins jQuery.

La librería jQuery, incorpora funciones que nos van a ayudar muchísimo, en la programación de nuestras aplicaciones. Además de todo lo que nos aporta la librería, disponemos de plugins o añadidos que aportan funcionalidades avanzadas.

Vamos a encontrar plugins en un montón de categorías: AJAX, animación y efectos, DOM, eventos, formularios, integración, media, navegación, tablas, utilidades, etc.

Para saber más

[Efectos con jQuery.](#)

[Documentación oficial de jQuery.](#)

Antes de poder usar cualquier plugin de jQuery, será necesario cargar primero la librería de jQuery, y a continuación la librería del plugin que deseemos, por ejemplo:

```
<script type="text/javascript" src="HTTP://code.jquery.com/jquery-latest.js"></script>
```

```
<script type="text/javascript" src="ejemploplugin.js"></script>
```

Todos los plugins contienen documentación, en la que se explica como usar el plugin.

Desde la web oficial de [jQuery](#), puedes hojear todos los plugins disponibles para jQuery:

[Plugins para jQuery.](#)

También es muy común el encontrar páginas, en las que se muestran rankings de los mejores plugins para jQuery. Algunos ejemplos pueden ser:

[Mejores plugins de jQuery del año 2011.](#)

[Los 130 mejores plugins de jQuery.](#)

[Búsqueda en Google de los mejores plugins de jQuery.](#)