Manual de AJAX

Las entrañas de AJAX

Basado en el PFC, “AJAX, Fundamentos y Aplicaciones.”
Escrito : Juan Mariano Fuentes
Dirigido: Sergio Gálvez Rojas

2ªEd 2009

Índice

Introducción

Antes de emprender la lectura de este texto hay ciertas cosas que sería bueno conocer para entender perfectamente que es AJAX y en qué lugar se ubica dentro del desarrollo de aplicaciones web. Por ello aprovecharemos esta introducción para hablar del entorno que rodea AJAX, de esta manera, cuando se comience el primer capítulo, el lector estará mejor preparado.

La Base de la Web Actual

Primero será bueno repasar las tecnologías estándar que de las que disponemos en todos los navegadores.

El lenguaje HTML (HyperText Markup Language)

HTML es el lenguaje básico con el que podemos crear páginas web, con el paso del tiempo se han ido añadiendo etiquetas a las que ya tenía además de dar soporte a otros lenguajes como CSS (Cascade Style Sheets).

Las versiones anteriores a la 3.2 del lenguaje están ya caducadas y no son útiles hoy día; hoy en día un nuevo proyecto se debería emprender intentando seguir el estándar XHTML que es la última versión, aprobada en enero 2000.

El lenguaje XHTML (eXtensible HTML)

Este lenguaje es el que ha supuesto el mayor cambio desde 1997 (HTML 3.2), ya que busca proporcionar páginas web ricas en contenido a un amplio abanico de dispositivos PC, Móviles y dispositivos con conexión inalámbrica.

XML (eXtensible Markup Language)

XML es un metalenguaje que fue ideado para “describir” un conjunto de datos como pudiera ser los campos de una tabla para intercambiar información de forma rápida y fácil. También permite especificar cómo tienen que ser los datos, por lo que se decidió especificar el lenguaje HTML con XML y nació XHTML.

CSS (Cascade Style Sheets), Hojas de estilo

En los primeros tiempos de las páginas web, se tenía en un mismo documento “.html” tanto los párrafos de texto e imágenes como su estilo, e indicábamos el tipo de atributos del texto dentro de las etiquetas HTML.

Ahora que tenemos las CSS, asignamos a las etiquetas una clase dentro del documento “.html”, y a esa clase contenida en otro documento le especificamos el formato, de esta forma tenemos dos documentos: uno con los datos y otro que dice cómo deben representarse.

Si quisiéramos hacer un cambio en la representación de una web compuesta por 100 páginas y las 100 leen el mismo CSS, con un solo cambio en el ".css" habríamos cambiado toda la representación automáticamente.

DOM (Document Object Model)

Cuando se carga una página web en el navegador, éste último crea asociado a la página una serie de objetos, de manera que mediante un lenguaje utilizable en la parte del navegador(el cliente), como JavaScript, pueden modificarse las características de esos objetos y con ello la página en sí.

Ya que la página se actualiza inmediatamente si realizamos algún cambio con JavaScript, se estamos hablado del término HTML dinámico: DHTML (Dynamic HTML).

Lenguajes de Cliente, (JavaScript)

Cuando el usuario ve una página web en su navegador, ésta puede tener, aparte del código HTML, código escrito en un lenguaje de script que se utiliza normalmente para dotar de dinamismo a las páginas, obteniendo DHTML.

El principal lenguaje utilizado hoy día es JavaScript; nació con Netscape 2.0 y la versión actual es la 1.9. Por su parte Microsoft también tiene otra implementación de JavaScript llamada Jscript con su versión 5.8.

Lenguajes de Servidor (JSP, PHP, Ruby, etc.)

A veces necesitamos enviar información al servidor y que éste la procese y nos responda (por ejemplo al conectarse a nuestra cuenta de correo), o que guarde información (por ejemplo en un foro). Para este procesamiento se utilizan los lenguajes de servidor PHP, JSP (el que utiliza este texto), etc. Puedes elegir el que más te guste con sus pros y sus contras, y su dinámica de funcionamiento es la siguiente:

Tenemos una página escrita en alguno de estos lenguajes guardada en el servidor; cuando un usuario (cliente) accede, el servidor la ejecuta y le devuelve el resultado al cliente, que será solo HTML, no contendrá lenguajes del lado del servidor ya que el navegador no los comprende.

Visión en Conjunto

Tendremos un usuario que carga en su navegador una página compuesta por XHTML y JavaScript cuyo estilo está en un archivo CSS si lo separamos; el navegador crea el DOM asociado a la página y el JavaScript lo modifica para conseguir dinamismo.

Tenemos en el servidor páginas hechas con JSP, cuando el usuario pide una de estas páginas, el servidor la ejecuta y devuelve el resultado al usuario ya sea una página XHTML u otro tipo de información.

Las RIA (Rich Internet Application Technologies)

Para que entendamos la necesidad de uso de AJAX, vamos a ver una serie de términos, problemas y posibles soluciones y ver cuál es el papel de AJAX dentro de todo esto.

Características de una aplicación de escritorio

Si le echamos un vistazo a una aplicación típica de escritorio vemos que tiene las siguientes cualidades.

- Responde de forma intuitiva y rápida.
- Da respuesta inmediata a los actos del usuario.

Características de una aplicación Web Convencional

- Cada vez que pulsamos sobre un link, esperamos y la página se refresca.
- La página refresca todos los eventos, envíos y datos de la navegación.
- El usuario debe esperar la respuesta.
- Modelo de petición/respuesta de comunicaciones síncrono.
- El estado del trabajo que estamos desarrollando se basa en qué página estamos.

Problemas de una aplicación web convencional

- Respuesta lenta.
- Pérdida del contexto durante el refresco.
- Perdemos información en la pantalla que habíamos rellenado.
- Perdemos la posición del scroll de la pantalla.
- No tenemos respuesta inmediata a nuestros actos.
- Tenemos que esperar que llegue la siguiente página.

Por estas razones nacieron las (RIA), Rich Internet Application Technologies.

- Applet
- Adobe Flash
- Java WebStart
- DHTML
- AJAX

Desglosemos cada una de las RIA para ver sus pros y sus contras.

Applet

Positivo

- Puede hacer uso de todas las APIS Java.
- Su desarrollo tiene un patrón de trabajo bien definido.
- Puede manipular gráficos, diferentes hebras y crear Interfaces Usuario avanzadas.

Negativo

- El navegador necesita un complemento.
- El tiempo de bajada del APPLET puede ser muy grande.

Adobe Flash

Fue diseñado para ver Películas Interactivas aunque ahora se utiliza mucho para hacer juegos.

Positivo

- Es capaz de dar un aspecto visual inigualable actualmente con otras tecnologías para una página web.
- Muy bueno para mostrar gráficos vectoriales 3D.

Negativo

- El navegador necesita un complemento.
- ActionScript es una tecnología propietaria.
- El tiempo de bajada del vídeo o programa suele ser muy grande (lo bonito se paga).

Java WebStart

Podemos decir que nos proporciona desde Internet una aplicación de escritorio.

Positivo

- Una vez cargado, nos da una experiencia similar a una aplicación de escritorio.
- Utiliza Tecnología muy extendida como Java.
- Las aplicaciones se pueden firmar digitalmente.
- Se pueden seguir utilizando una vez desconectado

Negativo

- El navegador necesita un complemento.
- Problema de compatibilidad con las aplicaciones viejas ya que se han cambiado algunas cosas.
- El tiempo que puede tardar en descargar una aplicación de escritorio es demasiado grande.

DHTML = HTML + Javascript + DOM + CSS

POSITIVO

- Se utiliza para crear aplicaciones interactivas y más rápidas.

NEGATIVO

- La comunicación es síncrona.
- Requiere el refresco completo de la página, perdiendo parte del contexto.

AJAX = DHTML + XMLHttpRequest

Es un refinamiento del DHTML, utiliza todas sus herramientas, sumándole el objeto XMLHttpRequest para obtener información de manera asíncrona y refrescar solo la parte necesaria de la página sin perder nada del contexto, se terminaron los problemas del DHTML.

Positivo

- La mejor tecnología RIA hasta el momento.
- Está en su mejor momento para la industria.
- No necesitamos descargar un complemento.

Negativo

- Todavía existen incompatibilidades entre navegadores (cada vez menos).

- Desarrollo con JavaScript, hace un par de años no muy explorado pero hoy en día con cierta consistencia.

Con todo lo anterior, vemos que hoy en día, una de las mejores posibilidades y más nueva para ofrecer una experiencia rica al usuario es la utilización de AJAX.

Algunos ejemplos reales de AJAX

Lo mejor es ver algunas posibilidades del uso de AJAX, se va a hacer mención primero a la primera aplicación AJAX conocida, Microsoft inventó el objeto que más tarde se convertiría en el XMLHttpRequest y lo usó en su versión web del Outlook (versión de 1998).

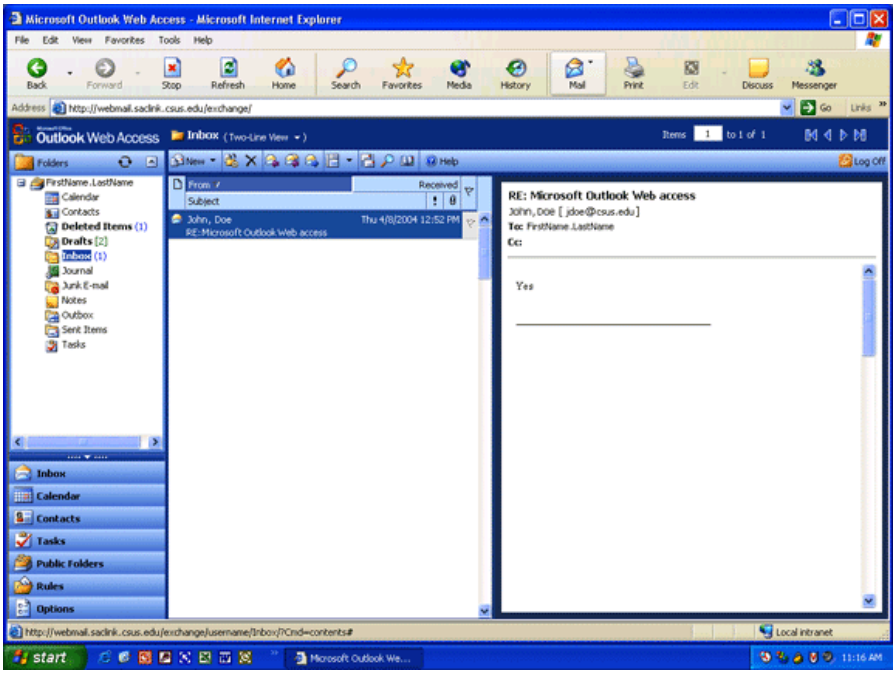


Ilustración 1 Outlook Web Access, imagen encontrada buscando en Google.

¿Cómo, es que no empezó su utilización masiva hasta el año 2005? Solo Internet Explorer era capaz de generar el objeto XMLHttpRequest(llamado de otro modo).

Cuando Mozilla Firefox, el navegador más grande de la competencia, implementó un objeto compatible, y más tarde el resto de navegadores de código abierto, empezó el boom.

Otro gran ejemplo del uso de AJAX es Google Maps (http://maps.google.com/), que es más conocido y la verdad llama muchísimo más la atención por la cantidad de zoom que se puede hacer en los mapas, convirtiéndose en una verdadera guía para no perdernos por la ciudad.

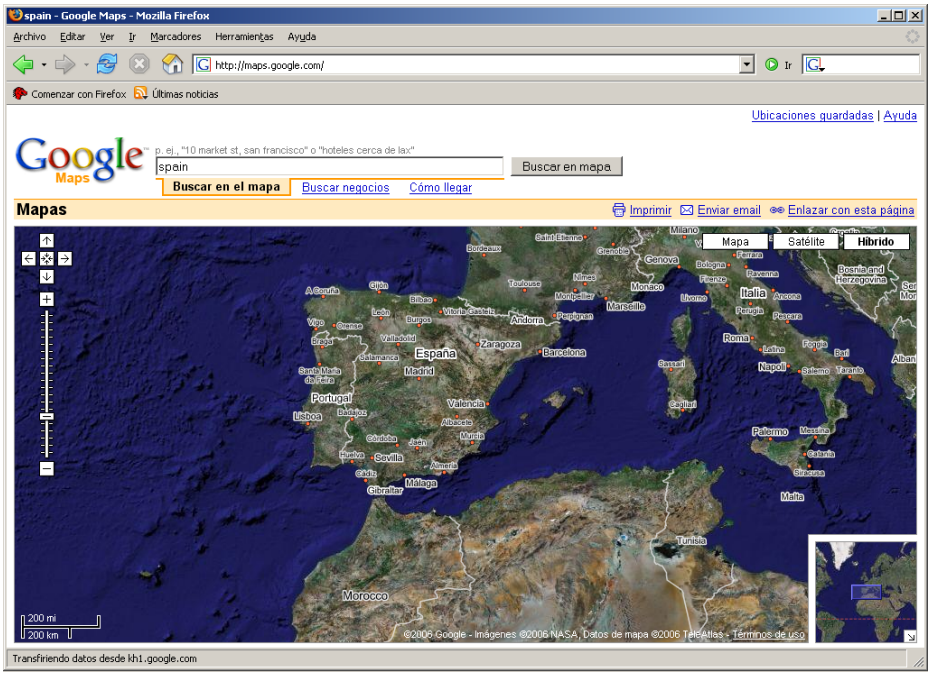


Ilustración 2 Google Maps, vista de España.

Notas finales

La documentación que sigue ha sido desarrollada basándose en muchas fuentes, se ha hecho intentando que la curva de aprendizaje sea lo menos pronunciada posible por si hay personas que se quieren iniciar en el desarrollo web con AJA

Capítulo 1: El objeto XMLHttpRequest

1.1 Descripción del capítulo.

Para comenzar se va a construir una pequeña librería programada haciendo uso de JavaScript, primero para comprender cómo funciona el objeto XMLHttpRequest que se ha hecho muy famoso ya que constituye las entrañas de AJAX y segundo porque es muy recomendable utilizar una librería en el lado cliente para ahorrarnos no sólo problemas, como veremos seguidamente, sino también para no repetir el mismo código continuamente.

Es muy recomendable que si el lector piensa probar y modificar el código fuente de los ejemplos (la mejor forma de aprender), instale Apache Tomcat, un contenedor web capacitado para ejecutar páginas JSP que son las que se utilizan en este libro.

Los apartados del capítulo con su descripción general son las siguientes:

1.2 Razones para utilizar una librería en el lado cliente
- Enumeraremos las razones para hacer la librería ya que sino no tiene mucho sentido hacerla.

1.3 La dependencia de los navegadores
- Veremos las diferencias a la hora de crear el objeto en los diferentes navegadores, pero también veremos que el modo de empleo es igual.

1.4 Navegadores compatibles
- Aunque el objeto no es 100% compatible con todos los navegadores debemos pensar que sí, en todas las últimas versiones de los navegadores actuales.

1.5 Métodos y propiedades del objeto
- En este apartado tendremos una visión general de todas las funciones y propiedades a las que podemos acceder y en qué momento tiene sentido utilizarlas.

1.6 Constructor del objeto XMLHttpRequest
- Haremos una función que construya el objeto compatible al navegador con el que estamos usando.

1.7 Peticiones síncronas y asíncronas
- Comprender qué diferencia hay entre ellas cuando se utilizan haciendo uso del objeto XMLHttpRequest y por qué sólo usaremos las asíncronas.

1.8 La clase petición AJAX

- Construiremos una clase que nos libere del trabajo repetitivo de crear ciertas funciones que son necesarias en cada petición.

1.9 Escribir clases en JavaScript

- Comprender el por qué está construida la clase petición AJAX de la forma que está, así como una visión de cómo se crean los prototipos (clases de JavaScript) y las diferencias de éstos con las clases de Java que son mas conocidas para que se comprenda bien tanto el código de la clase petición AJAX como el siguiente código que escribamos en JavaScript.

1.2 Razones para utilizar una librería en el lado cliente.

Características básicas que debe cubrir una librería en el lado del cliente.

- La tecnología que use el servidor debe ser indiferente, es decir no debe actuar directamente con tecnologías como PHP, JSP, ASP, etc.
- Debe ser accesible en cualquier momento, localmente a ser posible.
- Debe manejar las incompatibilidades de los navegadores y hacer el código compatible.
- Debe manejar la comunicación asíncrona ocultando las operaciones a bajo nivel.
- Debe dar al desarrollador una forma fácil de acceder al DOM.
- Debe dar cierto manejo ante errores, primero para que el programa no se rompa y segundo para proporcionar cierta información al desarrollador.
- Debería de intentar seguir una programación orientada a objetos y que éstos fueran reutilizables.

Como puede verse las librerías ahorrarán multitud de problemas, de otra manera intentar hacer una aplicación Web medianamente vistosa se convierte en un trabajo para chinos, que además será casi imposible de mantener.

Características avanzadas que podría cubrir una librería en el lado del cliente.

- Proporcionar diferentes objetos gráficamente agradables directamente, como calendarios, botones, ventanas desplegables, etc.
- Proporcionar interfaces de usuario avanzadas, con animaciones y diferentes efectos gráficos que hagan la experiencia más agradable.

Razones para utilizar una de las librerías avanzadas ya existentes en Internet.

- Son mejores que la tuya propia ya que están echas normalmente por multitud de desarrolladores.
- Establecen comunidades de forma que la comunidad le añade prestaciones y es fácil conseguir ayuda en sus foros.
- Los entornos IDE no tardarán mucho tiempo en darle soporte, al menos a las más importantes de código abierto.

La librería que construiremos cubrirá las características básicas; para cubrir las características avanzadas existen multitud de librerías de código abierto y comerciales, además no aportan demasiado a la comprensión del problema que resuelve AJAX sino más bien, como he dicho ya, mejoran la experiencia visual. Además muchas de ellas pueden ser difíciles de utilizar para el usuario no experto y las que pueden ser mas fáciles esconden totalmente el problema, las usaremos para trabajar mas fácilmente una vez sepamos utilizar AJAX a bajo nivel.

Si vamos a utilizar AJAX a bajo nivel lo primero que debemos aprender es como crear el objeto en los diferentes navegadores y las similitudes y diferencias que son mínimas al tratarlo en uno y otro, además de solucionarlas, así que empezaremos por aquí en el siguiente punto.

1.3 La dependencia de los navegadores.

Antes de empezar a trabajar debemos de saber que el objeto XMLHttpRequest no es estándar, fue originariamente inventado por Microsoft, usado desde Internet Explorer 5.0 como un objeto ActiveX, siendo accesible mediante JavaScript. Mozilla Firefox en su versión 1.0 implementó un objeto compatible.

Ya que estos dos son los navegadores mas difundidos y además hay navegadores basados en el código de Mozilla Firefox, intentaremos que lo que hagamos sea compatible en ambos, de esta manera conseguiremos que nuestro código funcione mas del 90% de las veces ya que estos navegadores abarcan el mercado.

Ejemplo 1 en Internet Explorer

Archivo "ejemploIE.html

```
<html><head><title>Página de ejemplo</title>
<script language="JavaScript" type="text/javascript">
var peticion01 = null; //Creamos la variable
//Para Internet Explorer creamos un objeto ActiveX
peticion01 = new ActiveXObject("Microsoft.XMLHTTP");
      function Coger(url) {//Función coger, en esta caso le entra
una dirección relativa al documento actual.
            if(peticion01) { //Si tenemos el objeto  peticion01
                  peticion01.open('GET', url, false); //Abrimos la
url, false=forma síncrona
                  peticion01.send(null); //No le enviamos datos al
servidor.
                  //Escribimos la respuesta en el campo con
ID=resultado
                  document.getElementById('resultado').innerHTML =
peticion01.responseText;
                  }
      }
</script>

</head>
<body>
<!--Cuando ocurra el evento oneclick se llamará la función coger-->
<button onclick="Coger('datos/videoclub.xml')">Coge un
documento</button>
<table border="4">
      <tr>
<!--El campo con id=resultado se sustituirá por causa de que ese id
```

```
está en la función coger-->
      <td><span id="resultado">Sin resultado</span></td>
      </tr>
</table>
</body></html>
```

Archivo "datos/videoclub.xml"

```
<?xml version="1.0" encoding="UTF-8"?>

<VideoClub>

<Pelicula>
      <Titulo>El rey Leon</Titulo>
      <Duracion>87 Min</Duracion>
      <Genero>Animacion infantil</Genero>
</Pelicula>

<Pelicula>
      <Titulo>Aladin</Titulo>
      <Duracion>86 Min</Duracion>
      <Genero>Animacion infantil</Genero>
</Pelicula>

</VideoClub>
```

Por ahora no usaré archivos XML con acentos, ya que estos caracteres saldrán sustituidos por un símbolo extraño y es un problema que se resolverá más tarde.

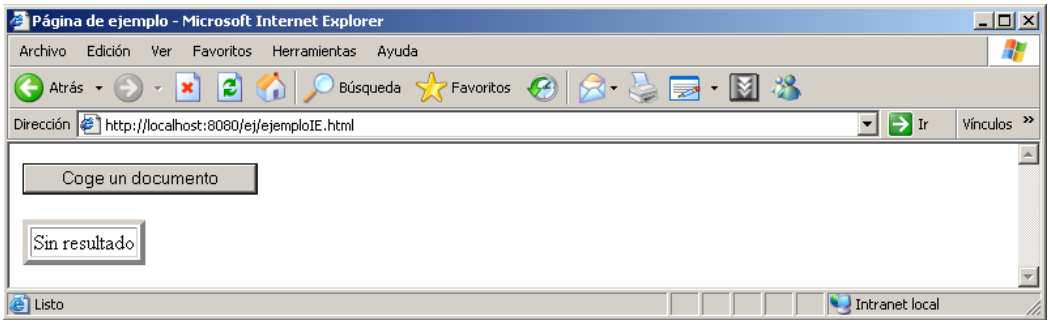


Ilustración 3 Primer ejemplo antes de pulsar el botón.

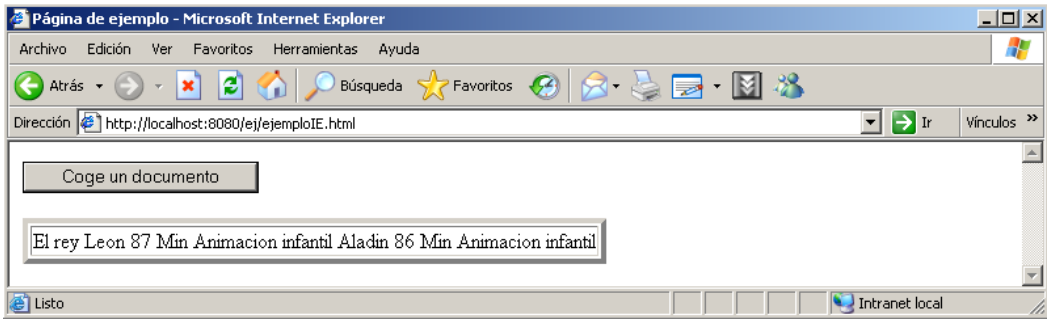


Ilustración 4 Primer ejemplo, después de recibir la información.

Como vemos se ha actualizado el campo de la página sin necesidad de refrescarla con el botón del navegador, pero este ejemplo no funcionaría en Mozilla Firefox, veamos los cambios que habrían en el código.

Ejemplo 1 en Mozilla Firefox

Lo único que tendremos que hacer será copiar el archivo "ejemploIE.html", renombrarlo "ejemploMF.html" y cambiar la línea:

```
peticion01 = new ActiveXObject("Microsoft.XMLHTTP");
```

Por la siguiente:

```
peticion01 = new XMLHttpRequest();
```

En los siguientes ejemplos, el código remarcado resalta qué cambia entre un ejemplo y el siguiente.

Con esto generamos en vez del Objeto ActiveX el XMLHttpRequest que es compatible con éste, al ser compatible no necesitamos cambiar mas líneas de código, veamos el ejemplo otra vez, funcionando ahora en Mozilla Firefox.

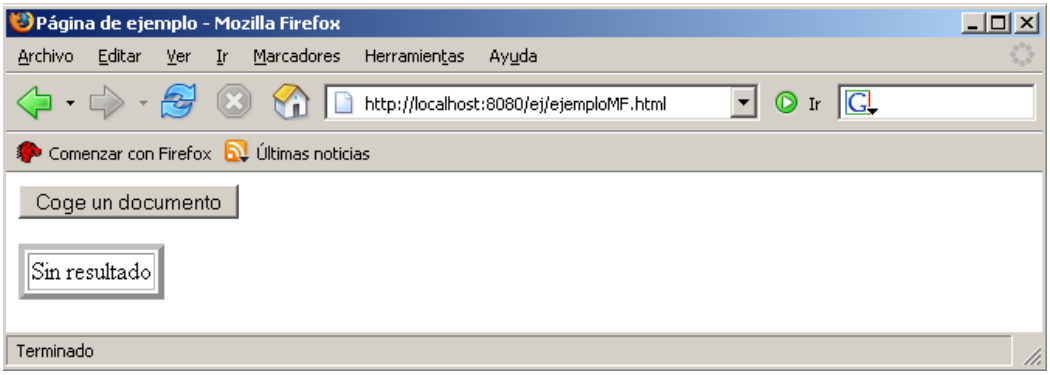


Ilustración 5 Primer ejemplo, ahora cargado en Mozilla Firefox.

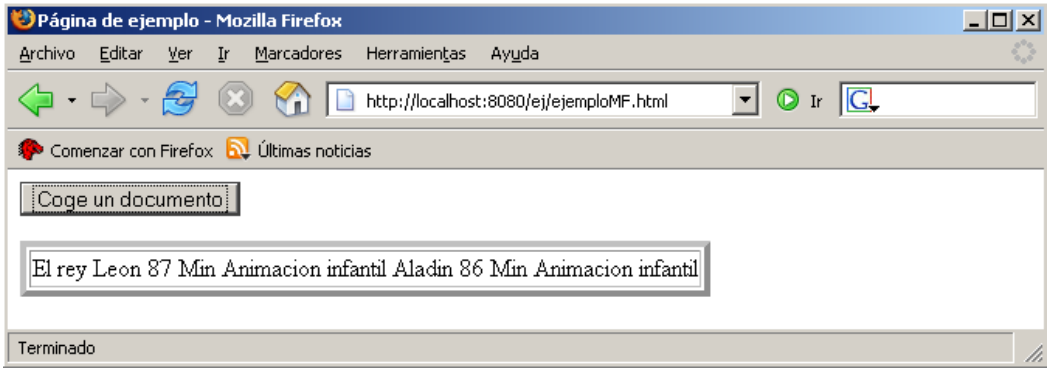


Ilustración 6 Primer ejemplo, mostrando el resultado en Mozilla Firefox.

Como hemos visto la diferencia en el código no es mas que cambiar el objeto que se crea, para que el ejemplo funcione en cualquier navegador bastaría con detectar el tipo de navegador y crear un objeto u otro; como esto es posible, es lo siguiente que haremos, así dejaremos de pensar en los navegadores, pero antes sería bueno que le echemos un vistazo a los navegadores compatibles y las propiedades del objeto, así terminaremos con la parte más teórica.

1.4 Navegadores compatibles.

Aunque nos centremos en Mozilla Firefox e Internet Explorer, la lista completa de los navegadores que actualmente soportan el objeto XMLHttpRequest es la siguiente:

Mozilla Firefox 1.0 y superiores
Netscape version 7.1 y superiores
Apple Safari 1.2 y superiores
Microsoft Internet Explorer 5 y superiores
Konqueror
Opera 7.6 y superiores

Como vemos esta muy extendido aunque no todos los navegadores lo soportan, ya que hay más, pero podemos decir que en ordenadores personales superamos el 95% de los posibles clientes, además, se está trabajando actualmente en navegadores para llevar AJAX hasta los dispositivos móviles.

1.5 Métodos y propiedades del objeto.

Métodos

Ya sea en Internet Explorer como en Mozilla Firefox, como hemos visto anteriormente, el objeto tiene una serie de métodos (funciones) que usamos para hacer la petición y se hace de igual manera en los dos navegadores.

Los métodos **open** y **send** son los que empleamos para establecer la conexión e iniciar la conexión, podemos decir que son los obligatorios y los que vamos a utilizar más.

Métodos	Descripción
open(método, URL, banderaAsync, nombreuser, password)	Según el método (GET o POST) y la URL se prepara la petición. Si la banderaAsync=true Petición asíncrona, si es fase la petición es síncrona. nombreuser y password solo se usan para acceder a recursos protegidos.
send(contenido)	Ejecuta la petición, donde la variable contenido son datos que se envían al servidor.
abort()	Para la petición que está procesando.
getAllResponseHeaders()	Devuelve todas las cabeceras de la llamada HTTP como un string.
getResponseHeader(cabecera)	Devuelve la cabecera identificada por la etiqueta.
setRequestHeader(etiqueta, valor)	Establece el valor de una etiqueta de las cabeceras de petición antes de que se haga la petición.

Propiedades

Además el objeto también tiene una serie de propiedades que cuando hacemos una petición de información nos indican cómo fue la petición (una vez terminada) y que normalmente utilizaremos de la siguiente manera.

```
//Código devuelto por el servidor, del tipo 404 (documento no
encontrado) o 200 (OK).
document.getElementById('estado').innerHTML = peticion01.status;
//Mensaje de texto enviado por el servidor junto al código (status),
para el caso de código 200 contendrá "OK".
document.getElementById('txtestado').innerHTML =
peticion01.statusText;
//Los datos devueltos por el servidor en forma de cadena.
document.getElementById('txtresultado').innerHTML =
peticion01.responseText;
//Datos devueltos por el servidor en forma de documento XML que
puede ser recorrido mediante las funciones del DOM
(getEementsByTagName, etc.).
document.getElementById('xmlresultado').innerHTML =
peticion01.responseXML;
```

Como vemos ahora, en el ejemplo anterior se ha utilizado el **responseText** para coger los datos del documento XML como texto (un string) y como hemos hecho

en el ejemplo anterior podemos añadir campos a la tabla con el **id** indicado y obtener el siguiente resultado.

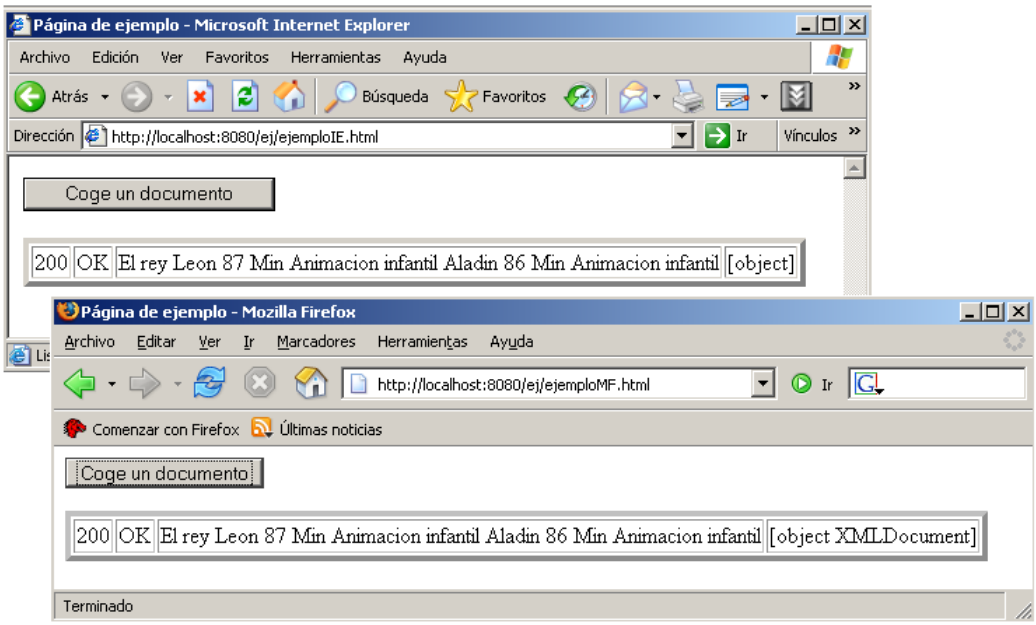


Ilustración 7 Propiedades del objeto mostradas en Internet Explorer y Mozilla Firefox.

En la ilustración anterior se tiene que las 3 primeras propiedades del objeto son iguales en los 2 navegadores, en cambio en la última dependiendo del navegador se muestra un texto diferente.

A todo lo anterior podemos sumar otras propiedades que podemos consultar mientras dura la petición para conocer su estado.

```
/*Sus valores varían desde 0(no iniciada) hasta 4(completado), en
cualquier caso tienes que hacer un switch, no puedes escribir su
valor directamente.*/
document.getElementById('estadoconexion').innerHTML =
peticion01.readyState;
/*Contiene el nombre de la función ejecutada cada vez que el estado
conexión cambia, es decir, nosotros asignamos una función que cada
vez que el estado dado por readyState cambia se lanza, por ejemplo
podríamos poner un gráfico cuando estemos en el estado de carga,
etc., como la anterior, no la puedes escribir directamente como
texto*/
document.getElementById('estadocambiante').innerHTML =
peticion01.onreadystatechange
```

Con lo cual, el cuadro resumen, de las propiedades del objeto seria el siguiente:

Propiedades	Descripción
status	Código devuelto por el servidor
statusText	Texto que acompaña al código
responseText	Datos devueltos formato string
responseXML	Datos devueltos formato Objeto XML
readyState	Estado actual de la petición. 0: Sin iniciar 1: Cargando

	2: Cargado 3: Interactivo (algunos datos devueltos) 4: Completado
onreadystatechange	Puntero a la función del manejador que se llama cuando cambia readyState.

Lo que hemos dicho aquí lo usaremos en los ejemplos siguientes ya que AJAX se basa en jugar con el objeto XMLHttpRequest como ya hemos dicho anteriormente.

1.6 Constructor del objeto XMLHttpRequest.

Llegados a este punto y como hemos dicho anteriormente, haremos una sencilla función que detectará el navegador dónde está funcionando la pagina web y según éste se creará el objeto de Internet Explorer o Mozilla Firefox o lo que es lo mismo el objeto ActiveX de Microsoft o el objeto XMLHttpRequest estándar que soportan el resto de los navegadores.

Los cambios en el código principal del ejemplo anterior son mínimos y están remarcados, lo único que vamos a hacer va a ser añadir una librería que contendrá una función a la que llamamos en vez de la función XMLHttpRequest() o ActiveXObject().

Archivo “ejemplo.html”

```
<html>
<head>
<title>Página de ejemplo</title>
<script language="javascript"
src="lib/ConstructorXMLHttpRequest.js"></script>
<script language="JavaScript" type="text/javascript">
var peticion01 = null; //Creamos la variable para el objeto
XMLHttpRequest
//Este ejemplo emplea un constructor, debería funcionar en cualquier
navegador.
peticion01 = new ConstructorXMLHttpRequest();
function Coger(url) //Función coger, en esta caso le entra una
dirección relativa al documento actual.
{
  if(peticion01) //Si tenemos el objeto peticion01
  {
  peticion01.open('GET', url, false); //Abrimos la url, false=forma
  síncrona
  peticion01.send(null); //No le enviamos datos al servidor
  //Escribimos la respuesta en el campo con ID=resultado
  document.getElementById('resultado').innerHTML =
  peticion01.responseText;
  }
}
</script>

</head>
<body>
<!--Cuando ocurra el evento oneclick se llamara la función coger-->
<button onclick="Coger('datos/videoclub.xml')">Coge un
documento</button>
<table border="4">
<tr>
<!--El campo con id=resultado se sustituirá por causa de que ese id
está en la función coger-->
<td><span id="resultado">Sin resultado</span></td>
</tr>
</table>
</body></html>
```

Esta es la parte más sencilla, ahora démosle un vistazo a la función interesante, detectaremos el navegador o más bien si tiene un método de creación XMLHttpRequest o ActiveXObject, estos se pueden comprobar si están en el objeto window que en JavaScript es el objeto más alto en la jerarquía del navegador.

Archivo “lib\ConstructorXMLHttpRequest.js”

```
function ConstructorXMLHttpRequest()
{
  if(window.XMLHttpRequest) /*Vemos si el objeto window (la base de
  la ventana del navegador) posee el método
  XMLHttpRequest(Navegadores como Mozilla y Safari). */
  {
    return new XMLHttpRequest(); //Si lo tiene, crearemos el objeto
    con este método.
  }
  else if(window.ActiveXObject) /*Sino tenía el método anterior,
  debería ser el Internet Exp. un navegador que emplea objetos
  ActiveX, lo mismo, miramos si tiene el método de creación. */
  {
  /*Hay diferentes versiones del objeto, creamos un array, que
  contiene los diferentes tipos desde la
  versión mas reciente, hasta la mas antigua */
  var versionesObj = new Array(
  'Msxml2.XMLHTTP.5.0',
  'Msxml2.XMLHTTP.4.0',
  'Msxml2.XMLHTTP.3.0',
  'Msxml2.XMLHTTP',
  'Microsoft.XMLHTTP');

    for (var i = 0; i < versionesObj.length; i++)
    {
      try
      {
      /*Intentamos devolver el objeto intentando crear las diferentes
      versiones se puede intentar crear uno que no existe y se
      producirá un error. */
      return new ActiveXObject(versionesObj[i]);
      }
      catch (errorControlado) //Capturamos el error, ya que podría
      crearse otro objeto.
      {
      }
    }
  }
/* Si el navegador llego aquí es porque no posee manera alguna de
crear el objeto, emitimos un mensaje de error. */
throw new Error("No se pudo crear el objeto XMLHttpRequest");
}
```

Para su mejor comprensión se ha comentado el código con mucha atención, si se lee tranquilamente se entiende perfectamente.

Ahora podemos dejar de preocuparnos por el navegador, y nos podemos centrar en utilizar el objeto correctamente.

1.7 Peticiones síncronas y asíncronas.

Si volvemos sobre nuestro ejemplo y nos fijamos en la siguiente línea:
peticion01.open('GET', url, false); //Abrimos la url, false=forma síncrona
Si pensamos un momento la técnica AJAX viene de Asíncrono, entonces ¿porqué estamos haciendo peticiones síncronas y que diferencia hay?, esto es lo siguiente que vamos a ver.

Si realizamos un petición síncrona el navegador queda bloqueado hasta que recibe la información, hasta ahora no lo hemos notado ya que estamos haciendo unas pruebas muy sencillas y no estamos recibiendo gran cantidad de información.

Para darnos cuenta de la situación vamos a pedir una página hecha en JSP que tiene una espera corta, seguimos basándonos en nuestro ejemplo, cambiando una línea:

Archivo "ejemplo.html"

```
<html><head><title>Página de ejemplo</title>
<script language="JavaScript" src="lib/ConstructorXMLHttpRequest.js">
</script>
<script language="JavaScript" type="text/javascript">
      var peticion01 = null; //Creamos la variable para el objeto
      peticion01 = new ConstructorXMLHttpRequest();

      function Coger(url) { //Función coger, en esta caso le entra
una dirección relativa al documento actual.
            if(peticion01) { //Si tenemos el objeto  peticion01
            peticion01.open('GET', url, false); //Abrimos la url,
false=forma síncrona
            peticion01.send(null); //No le enviamos datos al servidor
            //Escribimos la respuesta en el campo con ID=resultado
            document.getElementById('resultado').innerHTML =
peticion01.responseText;
            }
      }
</script>

</head>
<body>
<!--Cuando ocurra el evento oneclick se llamara la función coger-->
<button onclick="Coger('espera.jsp')">Coge un documento</button>
<table border="4">
      <tr>
<!--El campo con id=resultado se sustituirá por causa de que ese id
está en la función coger-->
      <td><span id="resultado">Sin resultado</span></td>
      </tr>
</table>
</body>
</html>
```

Archivo "espera.jsp"

```
<%
Thread.sleep(1000);
out.print("Es molesto tener que esperar de esta manera porque no
puedes hacer nada.");
%>
```

Aparte de esto, seguiremos utilizando el constructor que hemos hecho, tal como está, ejecutemos la página ver el resultado.

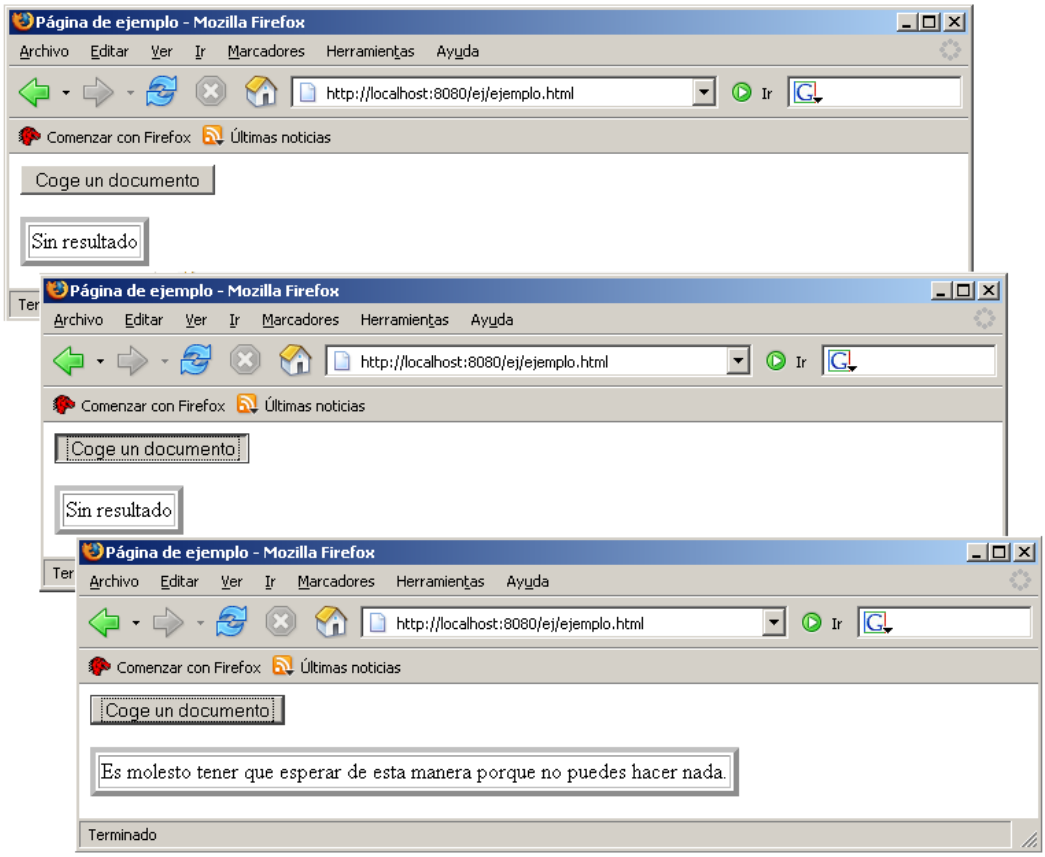


Ilustración 8 Problemas de una petición síncrona, mostrados en Mozilla Firefox. El botón permanece pulsado y la página bloqueada (no responde) hasta que se recibe la información pedida.

En cambio si realizamos una petición asíncrona el usuario queda libre de seguirse moviéndose por la página hasta que recibe la información, es decir, aumentamos la interactividad, en el ejemplo anterior, no nos quedaríamos con el botón pulsado esperando algo y podríamos seguir desplazándonos por la página (somos conscientes de que en este ejemplo no hay mucho por donde desplazarse), para esto bastaría con cambiar el false por un true:

```
peticion01.open('GET', url, true); //Abrimos la url, true=forma
asíncrona
```

El bloque de código JavaScript funciona igual, sólo que queda bloqueado en segundo plano a la espera de recibir el resultado de la petición pero sin bloquear al navegador; de ahora en adelante utilizaremos peticiones asíncronas para aumentar la interactividad.

Ahora surge un problema, ¿cómo sabe el usuario que está esperando la información, si no tiene muestra alguna de ello?, él sólo sabe que ha pulsado un botón (en nuestro ejemplo), lo ideal sería que recibiera alguna señal, bueno eso es lo que haremos en el siguiente ejemplo, de forma asíncrona.

Archivo “ejemplo.html”

```
<html><head><title>Página de ejemplo</title>
<script language="JavaScript" src="lib/ConstructorXMLHttpRequest.js">
</script>
<script language="JavaScript" type="text/javascript">
	var peticion01 = null;
	//Creamos la variable para el objeto xmlhttprequest
	peticion01 = new ConstructorXMLHttpRequest();
		//Se llama cuando cambia peticion01.readyState.
		function estadoPeticion() {
	switch(peticion01.readyState) { //Según el estado de la
petición devolvemos un Texto.
		case 0:  document.getElementById('estado').innerHTML =
"Sin iniciar";
		break;
		case 1: document.getElementById('estado').innerHTML =
"Cargando";
		break;
		case 2: document.getElementById('estado').innerHTML =
"Cargado";
		break;
		case 3: document.getElementById('estado').innerHTML =
"Interactivo";
		break;
		case 4: document.getElementById('estado').innerHTML =
"Completado";
		//Si ya hemos completado la petición, devolvemos además
la información.
		document.getElementById('resultado').innerHTML=
peticion01.responseText;
		break;
		}
	}
	function Coger(url) { //Función coger, en esta caso le entra
una dirección relativa al documento actual.
		if(peticion01) { //Si tenemos el objeto  peticion01
		peticion01.open('GET', url, true); //Abrimos la url,
true=forma asíncrona
		/*Asignamos la función que se llama cada vez que cambia
el estado de peticion01.readyState Y LO HACEMOS ANTES THE HACER EL
SEND porque inicia la transmisión.*/
		peticion01.onreadystatechange = estadoPeticion;
		peticion01.send(null); //No le enviamos datos a la página
que abrimos.
		}
	}
</script></head><body>
<!--Cuando ocurra el evento oneclick se llamará la función coger-->
<button onclick="Coger('espera.jsp')">Coge un documento</button>
<table border="4">
	<tr>
		<td><span id="estado">Estado petición</span></td> <!--
Campo para indicar el estado de la petición-->
	<td><span id="resultado">Sin resultado</span></td>
	</tr>
</table>
</body>
</html>
```

Además es bueno cambiar el texto del archivo JSP, por uno mas acorde.

Archivo “espera.jsp”

```
<%
Thread.sleep(1000);
out.print("Ahora la espera es menos molesta.");
%>
```

Probamos la página y el resultado que nos queda es:

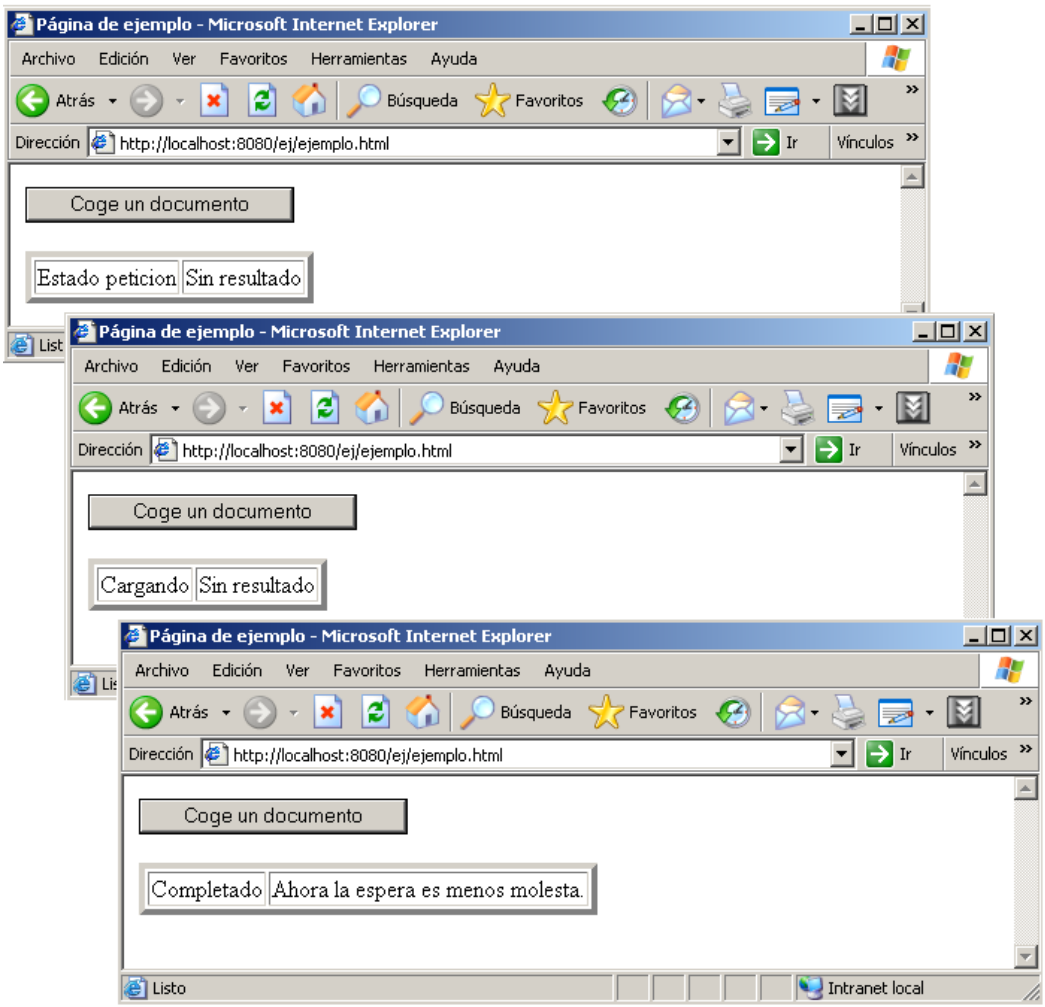


Ilustración 9 Una petición asíncrona, mostrada en Mozilla Firefox.

El resultado es satisfactorio, el texto que hemos puesto tampoco es que vaya a distraer mucho al usuario.

Cuando avancemos un poco más, podremos cargar imágenes que perfectamente pueden ser del tipo gif animado y mostrar así los diferentes estados de carga y error, quedando mucho más vistoso.

1.8 La clase petición AJAX.

Por ahora hemos hecho un recorrido que nos ha ayudado a comprender cómo utilizar el objeto **XMLHttpRequest** para pedir información de forma muy básica y hemos visto tanto sus métodos como propiedades.

Llegado aquí debemos fijarnos en el anterior ejemplo, para hacer una simple petición y poder controlar toda la información que nos da el objeto nos hemos visto obligados a hacer una función **coger(url)** y una **estadoPeticion()**, juntas son unas tres cuartas partes del documento. Vamos a construir una clase que contenga el objeto y sus funciones principales, de esta forma obtendremos un código más limpio y reutilizable, ya que tendremos una caja negra que funciona sin que nosotros tengamos que pensar demasiado (una vez que comprendamos lo que hace, por supuesto).

Primera aproximación (Errónea):
Como el objeto **XMLHttpRequest** no tiene un campo de identificador, vamos a añadírselo, de esta manera podremos reconocerlo(o eso podemos pensar), si tenemos varios simultáneamente.

Archivo "ejemploMalo.html"

```
<html>
<head>
<title>Página de ejemplo</title>
<script language="JavaScript"
src="lib/ConstructorXMLHttpRequest.js"></script>
<script language="JavaScript" type="text/javascript">
function estadoPeticion()
{
window.alert( "¿Que petición llego? (" + this.nombre + ")"); //Se
supone que  debería ser la del objeto al que se asignó.
}

function Coger(peticionAjax, url) //Le pusimos un campo mas, para
usarla con cualquier petición.
{
            if(peticionAjax){ //Hacemos la petición Ajax estándar
                  peticionAjax.open('GET', url, true);
                  peticionAjax.onreadystatechange = estadoPeticion;
                  peticionAjax.send(null);
            }
}
var Peticion01 = ConstructorXMLHttpRequest(); //Usamos el constructor
para obtener un objeto compatible.
Peticion01.nombre = "Peticion01"; //Un campo nombre, para saber quien
es.
window.alert( "Comprobando objeto 01 (nombre) = (" +
Peticion01.nombre + ")"); //Vemos que se le asigna el nombre
var Peticion02 = ConstructorXMLHttpRequest(); //Usamos el constructor
para obtener un objeto compatible.
Peticion02.nombre = "Peticion02"; //Un campo nombre, para saber quien
es.
window.alert( "Comprobando objeto 02 (nombre) = (" +
Peticion02.nombre + ")"); //Vemos que se le asigna el nombre
</script>
</head>
<body>
<!--Cuando ocurra el evento oneclick se llamara la función coger-->
<button onclick="Coger(Peticion01, 'datos/espera.jsp')">Coge el
documento 01</button>
<button onclick="Coger(Peticion02, 'datos/videoclub.xml')">Coge el
documento 02</button>
</body>
</html>
```

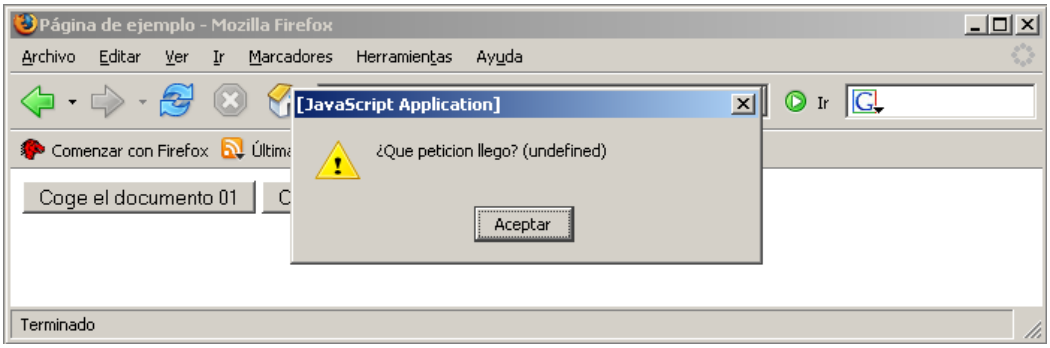


Ilustración 10 Fallo en la orientación a objetos de JavaScript, mostrada en Mozilla Firefox.

Pero para nuestra sorpresa (desagradable), esta solución tan sencilla nos daría el error que vemos en la ilustración anterior.

Esto ocurre porque cuando asignamos con el código:
```
peticionAjax.onreadystatechange = estadoPeticion;
```

No es que se copie la función estadoPeticion para el objeto de la petición actual, está utilizando la función global y nuestro objeto de petición no entra en la función de ninguna manera, sólo la dispara cuando cambia de estado y, por tanto, **this** no hace referencia al objeto **peticionAjax**. Este tipo de aproximaciones son inútiles debido a la orientación a objetos de JavaScript.

Segunda aproximación (La buena):
Ahora vamos a ver primero la clase que solucionaría el problema y luego el código principal, los he añadido en páginas separadas para que no quedara el código cortado, lo he comentado atentamente y lo he simplificado todo lo que he podido con el único ánimo de que pueda comprenderse leyéndolo, ya que por más que lo comente si no lo comprendes no hacemos nada. Léelo detenidamente, hasta ahora es lo más complejo que nos hemos encontrado en este texto y este código va a ser tu amigo, si comprendes bien este apartado, no tendrás problemas más adelante. Se recomienda echar un primer vistazo a este código y, posteriormente, aclarar conceptos en la siguiente sección. Por último, un nuevo vistazo al código resolverá el resto de las dudas.

Archivo "lib\ClasePeticionAjax.js"
```
/* El objetivo de este fichero es crear la clase objetoAjax (en
JavaScript a las "clases" se les llama "prototipos") */
function objetoAjax( ) {
/*Primero necesitamos un objeto XMLHttpRequest que cogeremos del
constructor para que sea compatible con la mayoría de navegadores
posible. */
this.objetoRequest = new ConstructorXMLHttpRequest();
}

function peticionAsincrona(url) { //Función asignada al método coger
del objetoAjax.
      /*Copiamos el objeto actual, si usamos this dentro de la
función que asignemos a  onreadystatechange, no funcionará.*/
      var objetoActual = this;
      this.objetoRequest.open('GET', url, true); //Preparamos la
conexión.
      /*Aquí no solo le asignamos el nombre de la función, sino la
función completa, así cada vez que se cree un nuevo objetoAjax se
asignara una nueva función. */
      this.objetoRequest.onreadystatechange =
            function() {
                  switch(objetoActual.objetoRequest.readyState) {
                        case 1: objetoActual.cargando();
                        break;
                        case 2: objetoActual.cargado();
                        break;
                        case 3: objetoActual.interactivo();
                        break;
                        case 4: /* Función que se llama cuando se
completo la transmisión, se le envían 4 parámetros.*/
      objetoActual.completado(objetoActual.objetoRequest.status,
                                    objetoActual.objetoRequest.statusText,
                              objetoActual.objetoRequest.responseText,
                            objetoActual.objetoRequest.responseXML);
                        break;
```

```
                }
            }
        this.objetoRequest.send(null); //Iniciamos la transmisión de
datos.
}
/*Las siguientes funciones las dejo en blanco ya que las
redefiniremos según nuestra necesidad haciéndolas muy sencillas o
complejas dentro de la página o omitiéndolas cuando no son
necesarias.*/
function objetoRequestCargando() {}
function objetoRequestCargado() {}
function objetoRequestInteractivo() {}
function objetoRequestCompletado(estado, estadoTexto,
respuestaTexto, respuestaXML) {}

/* Por último diremos que las funciones que hemos creado, pertenecen
al ObjetoAJAX,  con prototype, de esta manera todos los objetoAjax
que se creen, lo harán conteniendo estas funciones en ellos*/

//Definimos la función de recoger información.
objetoAjax.prototype.coger = peticionAsincrona ;
//Definimos una serie de funciones que sería posible utilizar y las
dejamos en blanco en esta clase.
objetoAjax.prototype.cargando = objetoRequestCargando ;
objetoAjax.prototype.cargado = objetoRequestCargado ;
objetoAjax.prototype.interactivo = objetoRequestInteractivo ;
objetoAjax.prototype.completado = objetoRequestCompletado ;
```

Archivo “ejemploBueno.html”

```
<html><head><title>Página de ejemplo</title>
<script language="JavaScript"
src="lib/ConstructorXMLHttpRequest.js"></script>
<script language="JavaScript"
src="lib/ClasePeticionAjax.js"></script>
<script language="JavaScript" type="text/javascript">
var PeticionAjax01 = new objetoAjax(); //Definimos un nuevo
objetoAjax.
PeticionAjax01.completado = objetoRequestCompletado01; //Función
completado del objetoAjax redefinida.
function objetoRequestCompletado01(estado, estadoTexto,
respuestaTexto, respuestaXML){
/* En el ejemplo vamos a utilizar todos los parámetros para ver como
queda, en un futuro próximo, solo te interesare la respuesta en
texto o XML */
document.getElementById('estado01').innerHTML = estado;
document.getElementById('estadoTexto01').innerHTML = estadoTexto;
document.getElementById('respuestaTexto01').innerHTML =
respuestaTexto;
document.getElementById('respuestaXML01').innerHTML = respuestaXML;
}

var PeticionAjax02 = new objetoAjax(); //Definimos un nuevo
objetoAjax.
PeticionAjax02.completado = objetoRequestCompletado02; //Función
completado del objetoAjax redefinida.
function objetoRequestCompletado02(estado, estadoTexto,
respuestaTexto, respuestaXML) {
/* En el ejemplo vamos a utilizar todos los parámetros para ver como
queda,
en un futuro próximo, solo te interesare la respuesta en texto o XML
*/
document.getElementById('estado02').innerHTML = estado;
document.getElementById('estadoTexto02').innerHTML = estadoTexto;
document.getElementById('respuestaTexto02').innerHTML =
respuestaTexto;
document.getElementById('respuestaXML02').innerHTML = respuestaXML;
}
</script>
</head>
```

```
<body>
<!--Cuando ocurra el evento oneclick se llamará a la función coger
DE CADA OBJETO! -->
<button onclick="PeticionAjax01.coger('datos/espera.jsp')">Coge el
documento 01</button>
<button onclick="PeticionAjax02.coger('datos/videoclub.xml')">Coge
el documento 02</button>
<table border="4">
      <tr>
          <td>Documento 01</td>
                  <!--Todos los campos del documento 01 les hemos
asignado un id como de costumbre para recibir la información -->
            <td><span id="estado01">estado no recibido</span></td>
            <td><span id="estadoTexto01">texto estado no
recibido</span></td>
            <td><span id="respuestaTexto01">texto respuesta no
recibido</span></td>
            <td><span id="respuestaXML01">xml respuesta no
recibido</span></td></tr>
      </tr>

      <tr>
      <td>Documento 02</td>
      <!--Todos los campos del documento 02 les hemos asignado un id
como de costumbre para recibir la información -->
      <td><span id="estado02">estado no recibido</span></td>
      <td><span id="estadoTexto02">texto estado no
recibido</span></td>
      <td><span id="respuestaTexto02">texto respuesta no
recibido</span></td>
      <td><span id="respuestaXML02">xml respuesta no
recibido</span></td></tr>
      </tr>
</table>
</body></html>
```

Como puedes ver en el código de la página principal, seguimos reutilizando los archivos espera.jsp y videoclub.xml colocándolos en una carpeta llamada datos, ahora veamos como queda el ejemplo completo, primero pediremos el primer documento y luego el segundo.

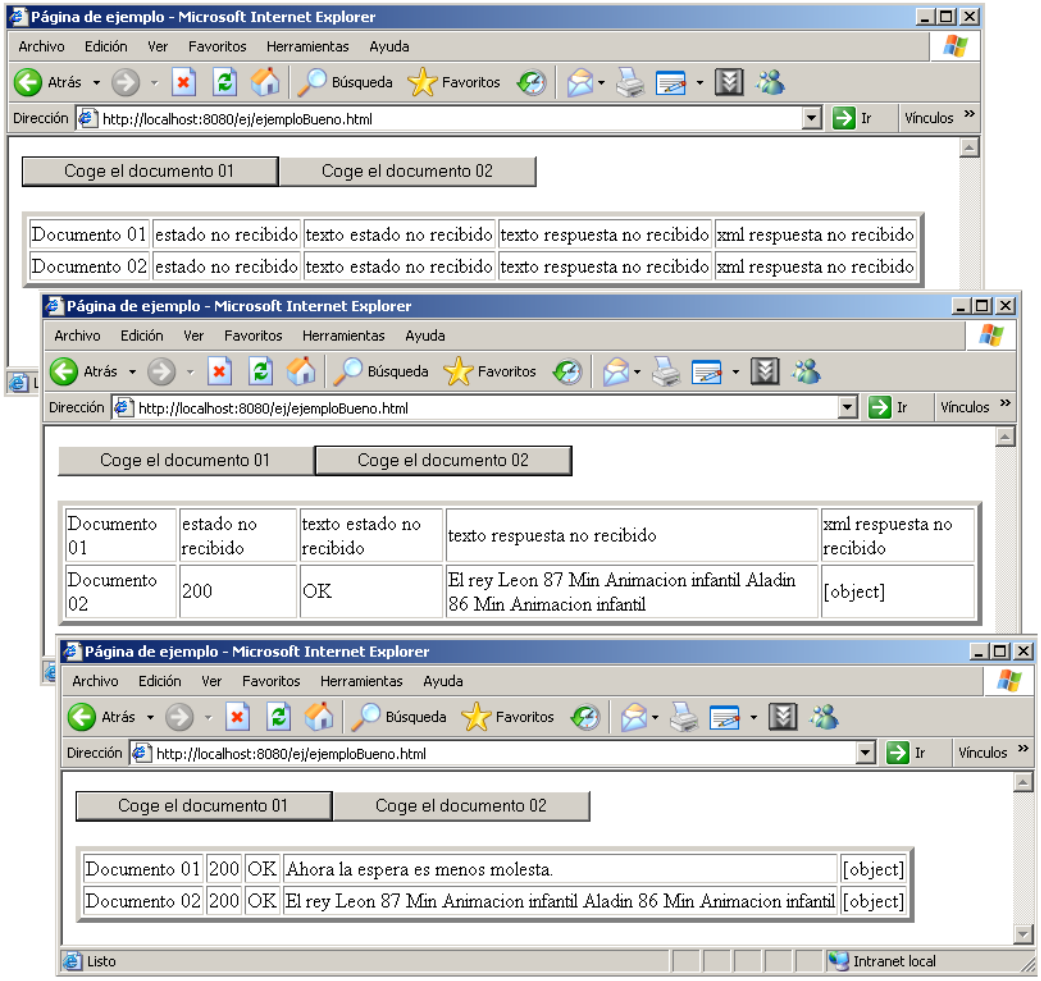


Ilustración 11 Resultado correcto a la hora de usar las clases construidas.

A causa de la espera, llega el segundo objeto antes que el primero y como lo hemos hecho de forma asíncrona no hemos tenido que esperar para poder pulsar el segundo botón, si hemos llegado hasta aquí comprendiéndolo todo, estamos listos para empezar a cargar de diferentes formas, HTML, imágenes, JavaScript, etc. , empleando AJAX que es nuestro objetivo.

1.9 Escribir clases en JavaScript.

Aun sin ser un lenguaje ni mucho menos raro, JavaScript no esta altamente difundido ni se ha usado demasiado hasta ahora, DHTML no tuvo mucho éxito, es ahora cuando ha surgido AJAX cuando se está viendo por parte de los desarrolladores ciertas medidas para escribir código más limpio y comprensible.

1.9.1 Clases VS Prototipos.

Ahora que hemos visto los primeros ejemplos vamos a comparar JavaScript con Java que seguramente conocemos más para ver las diferencias y tomar ciertas medidas de ahora en adelante para escribir código, Java es un lenguaje basado en clases como muchos sabremos pero JavaScript es un lenguaje basado en prototipos.

Basado clases (Java)	Basado prototipos (JavaScript)
Clases e instancias son entidades diferentes.	Todos los objetos son instancias.

Defines una clase con una definición de clase, instancias una clase con su constructor.	Defines y creas los objetos con los constructores.
Creas un nuevo objeto con el operador “new”.	Igual
Construyes una jerarquía de objetos usando definiciones de clases para definir subclases.	Construyes una jerarquía de objetos asignando un objeto como el prototipo asociado a una función constructor.
Heredan las propiedades siguiendo una cadena de clases.	Heredan las propiedades usando una cadena de prototipos.
Una definición de clase describe “todas” las propiedades de “todas” las instancias de la clase, no se pueden añadir dinámicamente.	La función constructora o prototipo especifica el conjunto de propiedades inicial, luego se pueden añadir mas ya sea a unos pocos o todos los objetos creados.
Traducido de “Core JavaScript Guide” de Netscape Communications Corp.	

Con el siguiente ejemplo se va a ver muchísimo mas claro:

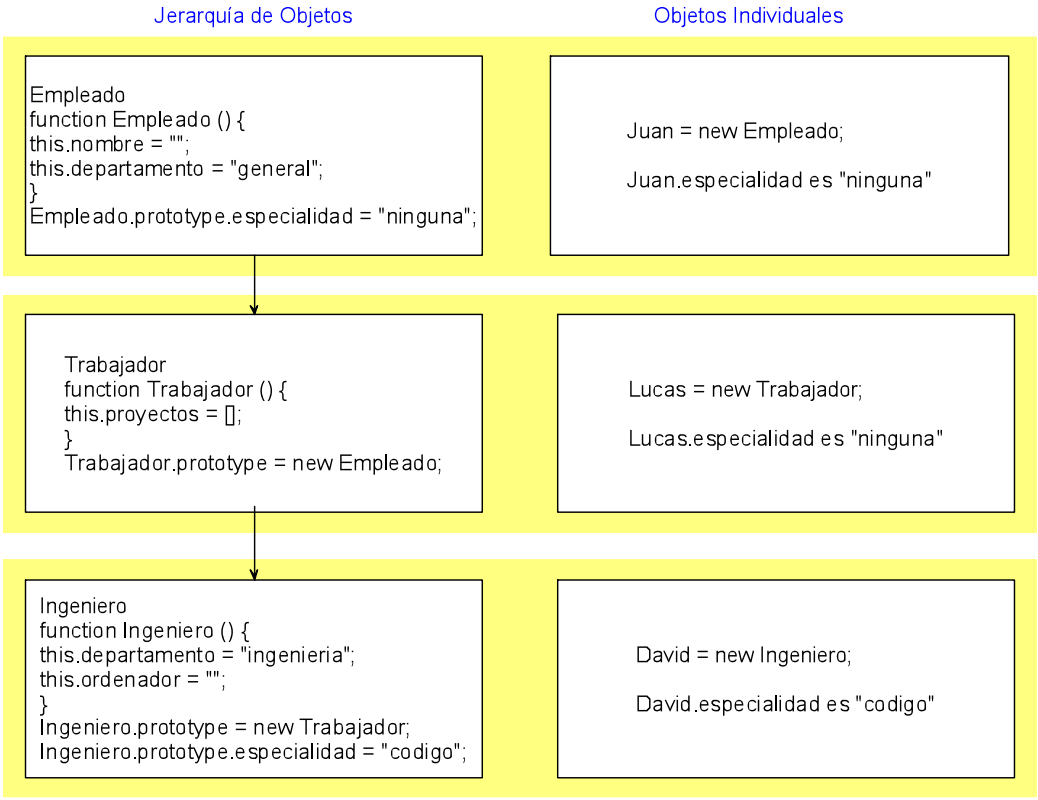


Ilustración 12 Ejemplo de uso de prototipos, traducido de “Core JavaScript Guide”.

1.9.2 Prototype VS encerrar las funciones.

Por supuesto todo lo que viene ahora, está visto y razonado desde el punto de vista de quien escribe estas líneas y no representan ningún estándar, otros autores pueden presentar sus propias alegaciones y recomendar justamente lo contrario.

Ahora se ve mejor porqué hemos utilizado la palabra reservada **prototype** para construir la clase anterior, vamos a discutir este tema, porque podríamos haberlo hecho sin usarla y el código quedaría incluso más claro.

Otra forma de escribir la ClasePeticionAjax sería la siguiente:

ClasePeticionAjaxV2.js

```
function objetoAjax( )
{
/*Primero necesitamos un objeto XMLHttpRequest que cogeremos del
constructor para que sea compatible con la mayoría de navegadores
posible. */
this.objetoRequest = new ConstructorXMLHttpRequest();
//Definimos la función de recoger información.
this.coger = function peticionAsincrona(url) //Función asignada al
método coger del objetoAjax.
{
    /*Copiamos el objeto actual, si usamos this dentro de la función
que asignemos a  onreadystatechange, no funcionara.*/
    var objetoActual = this;
    this.objetoRequest.open('GET', url, true); //Preparamos la
conexión.
    /*Aquí no solo le asignamos el nombre de la función, sino la
función completa, así cada vez que se cree un nuevo objetoAjax se
asignara una nueva función. */
    this.objetoRequest.onreadystatechange =
        function() {
            switch(objetoActual.objetoRequest.readyState)
            {
                case 1: //Función que se llama cuando se está
cargando.
                objetoActual.cargando();
                break;
                case 2: //Función que se llama cuando se a cargado.
                objetoActual.cargado();
                break;
                case 3: //Función que se llama cuando se está en
interactivo.
                objetoActual.interactivo();
                break;
                case 4: /* Función que se llama cuando se completo
la transmisión, se le envían 4 parámetros. */

    objetoActual.completado(objetoActual.objetoRequest.status,
                            objetoActual.objetoRequest.statusText,
                           objetoActual.objetoRequest.responseText,
                          objetoActual.objetoRequest.responseXML);
                break;
            }
        }
    this.objetoRequest.send(null); //Iniciamos la transmisión de
datos.
}
//Definimos una serie de funciones que sería posible utilizar y las
dejamos en blanco.
    this.cargando = function objetoRequestCargando() {}
    this.cargado = function objetoRequestCargado() {}
    this.interactivo = function objetoRequestInteractivo() {}
    this.completado    =    function    objetoRequestCompletado(estado,
estadoTexto, respuestaTexto, respuestaXML) {}
}
```

Vemos que todo queda encerrado dentro del mismo constructor, y sólo hacemos referencia a una función declarada fuera que construye el objeto XMLHttpRequest y que podríamos tener también dentro, de manera que el objeto se auto contendría.

Aunque pueda parecer bonito encerrar las funciones en el constructor y es una buena técnica para limpiar código que podemos recomendar, tiene un problema técnico. Encerrar las funciones puede ser ineficiente desde el punto de vista del

rendimiento y memoria, cada vez que hay una función encerrada en el constructor ésta se crea para cada objeto, lo que no es un problema si vamos a crear pocos objetos a la vez en un PC de hoy en día.

Pero, ¿qué pasa con otros terminales mas desfavorecidos como los diferentes terminales móviles que en un futuro podrían soportar el objeto XMLHttpRequest?, ya que no nos cuesta nada mejorar el rendimiento en esta ocasión, dejemos la función constructora como estaba antes.

1.9.3 Variables públicas VS variables privadas.

Los métodos de ClasePeticionAjax son todos públicos y la variable XMLHttpRequest también, podemos sentir la tentación en algunas ocasiones de declararlos como privados para que el desarrollador final que utilice nuestra clase no pueda tocarlos en el transcurso del programa y por consiguiente estropear algo de manera que tuviera que crear un nuevo objeto para cada petición (como en JavaScript no existe un public y un private, tendríamos que poner esas variables y métodos en un lugar donde no tuviera visibilidad el creador del objeto), a cambio estamos perdiendo mucha funcionalidad ya que en ocasiones será necesario reutilizarlo para facilitar la creación de algún programa o debido a que la memoria en terminales pequeños no es gratuita; por ello en principio las variables y métodos de los objetos que creemos serán públicos debido a las necesidades de trabajo que pueden surgir con AJAX.

Capítulo 2: Herramientas de depuración

2.1 Descripción del capítulo.

En el capítulo anterior se ha leído bastante código, si el lector en algún momento ha intentado hacer modificaciones puede que el navegador le diera algún error y que este no le fuera muy comprensible, además el objetivo de este texto es que el lector sea capaz de manejarse con la técnica y ya que principalmente se trabajará con JavaScript dentro de un entorno Web, este entorno debemos aprender a depurarlo, para ello vamos a ver en este capítulo las herramientas necesarias para ello. Puede que hayan más herramientas; se han escogido con el criterio de que sean tanto gratuitas como potentes, además tenemos la suerte de que las que vamos a ver se integran en un mismo programa, el navegador Mozilla Firefox.

2.2 Instalación.

Todo lo que necesitamos para la instalación es lo siguiente:

- Paquete instalación de Mozilla Firefox : Lo podemos bajar de la página principal de Mozilla Firefox que es http://www.mozilla.com/firefox/ .
- Paquete xpi de FireBug : (complemento que añade funcionalidades), se puede bajar de la web de complementos de Firefox que es https://addons.mozilla.org/firefox .

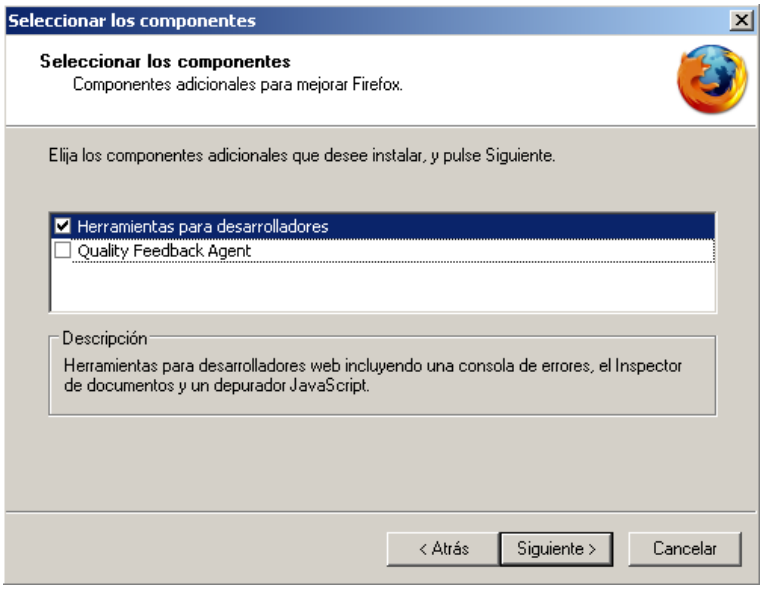


Ilustración 13 Cuadro selección componentes de la instalación personalizada de Mozilla Firefox.

Vamos a empezar instalando Mozilla Firefox, cuando hagamos las instalación debemos elegir hacer la instalación personalizada, elegiremos instalar las herramientas para desarrolladores.

Una vez que esté instalado Mozilla Firefox deberemos instalar el complemento. Para esto solo tenemos que ir a la barra de herramientas, archivo->Abrir archivo y seleccionamos el archivo si lo habíamos bajado antes, Firefox detectará que es una de sus extensiones y te preguntará si quieres instalarlo, reiniciamos Firefox y ya lo tendremos instalado.

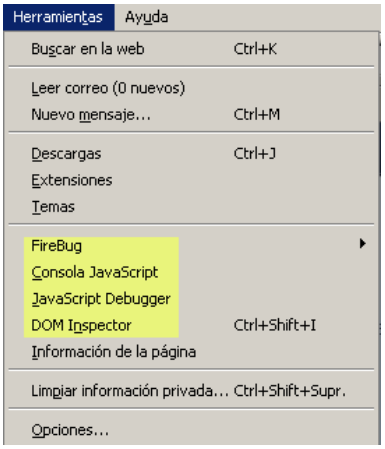


Ilustración 14 Aspecto de la ventana de herramientas tras la instalación de las diferentes utilidades de desarrollo.

Una vez que terminemos deberíamos tener las siguientes utilidades al iniciar Mozilla Firefox si miramos en la pestaña herramientas.

Resumiendo contamos con las siguientes utilidades (las menciono en orden en el que las vamos a ver).

- Consola JavaScript.
- DOM Inspector
- JavaScript Debugger (Venkman)

- FireBug

2.3 La consola JavaScript.

Para ver el funcionamiento de la consola vamos a hacer uso del siguiente ejemplo.

depuracion1.html

```
<html>
<head><title>depuración</title></head>
<script language="JavaScript" type="text/javascript">
  var nombre = "Juan" ;
  function hagoalgo()
  {
    longitud = nombre.length(); //Aquí hemos introducido un error.
    alert("La longitud del nombre es : " + longitud);
  }
</script>
<body>
PÁGINA PARA HACER PRUEBAS DE DEPURACIÓN SIMPLES
<br>
<a href="javascript:hagoalgo();">Llamo Función hagoalgo</a>
</body>
</html>
```

Si cargamos la siguiente página y si sacamos la consola y pinchamos en el link que hemos puesto a la función obtendremos lo siguiente.

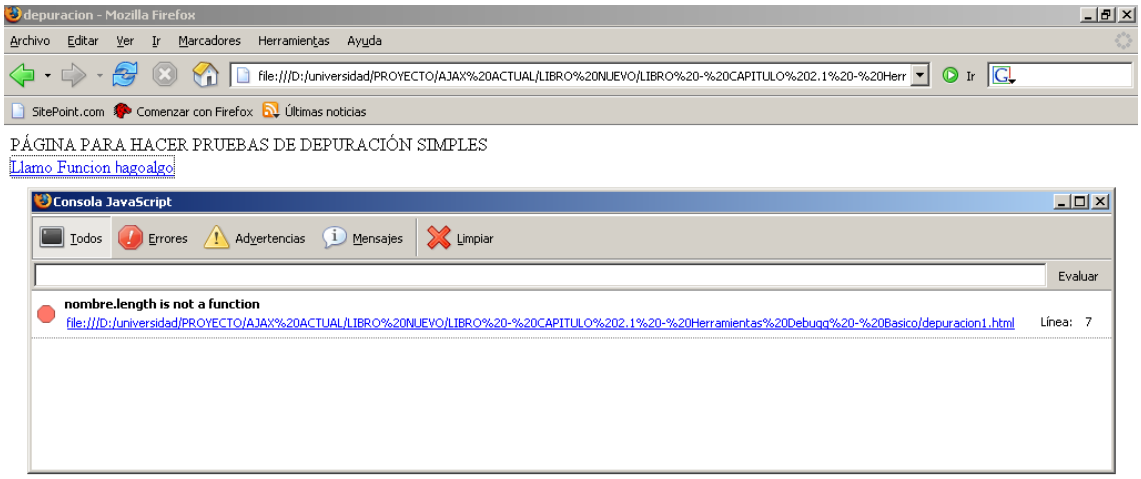


Ilustración 15 Resultado en la consola de hacer clic sobre el link en la página web.

Como vemos la propia consola se ha dado cuenta de que los objetos String no tienen una función llamada length. Si pulsamos encima del link de la consola nos llevará al punto del código que produjo el error.

Archivo Editar Ver Ayuda

```
<html>
<head><title>depuración</title></head>
<script language="JavaScript" type="text/javascript">
var nombre = "Juan" ;
function hagoalgo()
{
longitud = nombre.length(); //Aquí hemos introducido un error.
alert("La longitud del nombre es : " + longitud);
}
</script>
<body>
PÁGINA PARA HACER PRUEBAS DE DEPURACIÓN SIMPLES
<br>
<a href="javascript:hagoalgo();">Llamo Funcion hagoalgo</a>
</body>
</html>
```

Ilustración 16 Código que produjo un error mostrado por el propio Firefox.

Es muy recomendable combinar el uso de la consola con mensajes alert para depurar programas muy pequeños, cuando JavaScript encuentra un error como el anterior, el programa se rompe y no continua, si nos fijamos vemos que había puesto una función alert detrás de la línea donde se encuentra el error y no se ha ejecutado, así se puede estar seguro de que el programa no llega a ejecutar esa línea, además de que se puede monitorizar antes y después del error el valor que puede tener una variable problemática, es sencillo y esta herramienta no tiene mas complicación.

Solución del error anterior, length no es una función sino una propiedad; con quitar de la línea los paréntesis de la función "length()" y dejarlo en "length" se solucionaría.

2.4 Document Object Model inspector (inspector del DOM).

Esta herramienta nos permite ver el árbol de objetos del documento con todos los campos de cada etiqueta, su utilidad es la siguiente.

Imagínate que hemos hecho una inserción de código HTML en la página mediante el objeto XMLHttpRequest, pero dicho cambio no es apreciable y la consola no nos da ningún error.

Esto es porque el código insertado puede ser correcto pero no hace lo que debería, la solución es ver qué hemos insertado realmente mediante el inspector del DOM para buscar el error luego dentro del código que hemos insertado, veámoslo con el siguiente ejemplo.

depuracion2.html

```
<html>
<head><title>depuración</title></head>
<script language="JavaScript" type="text/javascript">
  function insertoalgo()
```

```
  {
    zona = document.getElementById('zonaInsercion') ;
    zona.innerHTML =  "<center><img src= \"hamsters.jpg\" idth=\"320\"
    eight=\"240\" border=\"0\" alt=\"Hamsters\"/></center>" ; //Aquí
    hemos introducido un error, en el alto y ancho, le hemos quitado
    una letra.
  }
</script>
<body>
PÁGINA PARA HACER PRUEBAS DE DEPURACIÓN SIMPLES
<br>
      <span id="zonaInsercion">
            <a href="javascript:insertoalgo();">Llamo Función
insertoalgo</a>
      </span>
</body>
</html>
```

Vamos a insertar directamente una imagen, puesto que sabemos donde está y no hacemos ninguna petición al servidor esto no es AJAX, pero puede sernos útil alguna vez.

Nuestra imagen es una imagen con una resolución 1024x768 y pensamos insertar en un espacio 320x240, si abrimos la página y pinchamos sobre el link que dispara la petición obtenemos el siguiente resultado.

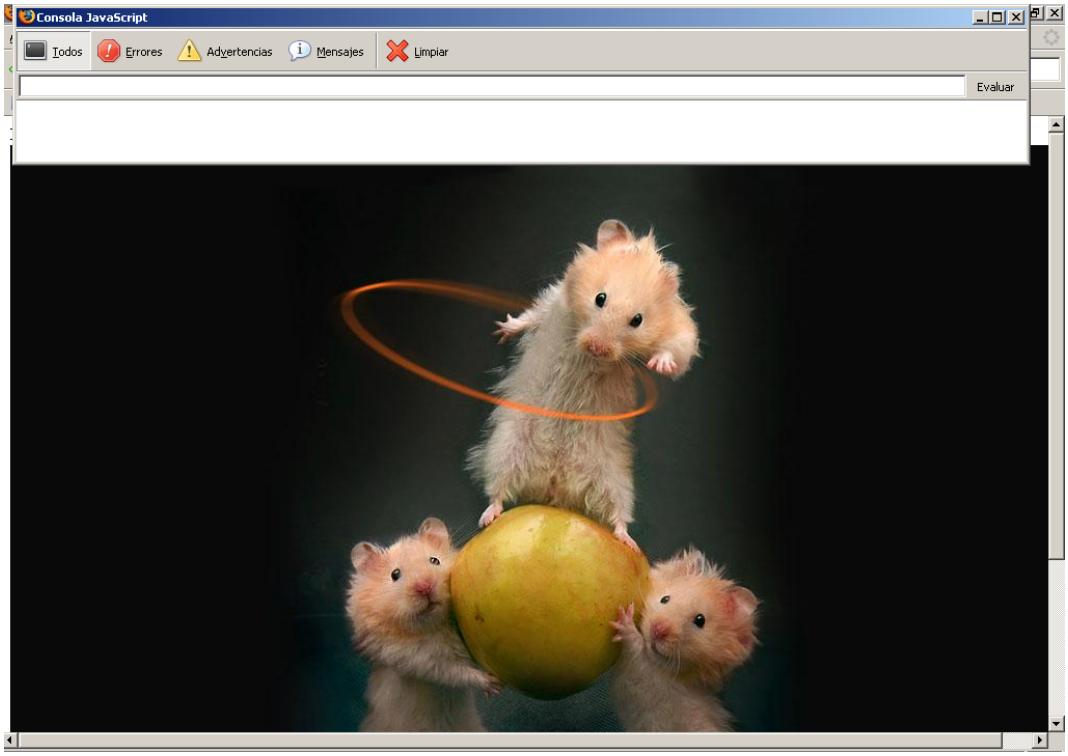


Ilustración 17 Resultado del código depuracion2.html visto en Mozilla Firefox.

Como vemos no hay ningún error en la consola JavaScript y la imagen se ha insertado con su tamaño original, no respetando las medidas que pretendíamos, el error se encuentra en el código HTML, este ejemplo es sencillo y fácil, pero la primera vez que nos pasa puede llevarnos de cabeza, con el DOM inspector podemos ver qué propiedades se han insertado correctamente y cuáles no.

Abrimos pues la ventana del DOM y abrimos las sucesivas pestañas del árbol jerárquico que forman las etiquetas HTML como muestra la figura 2.6 hasta llegar a la imagen.

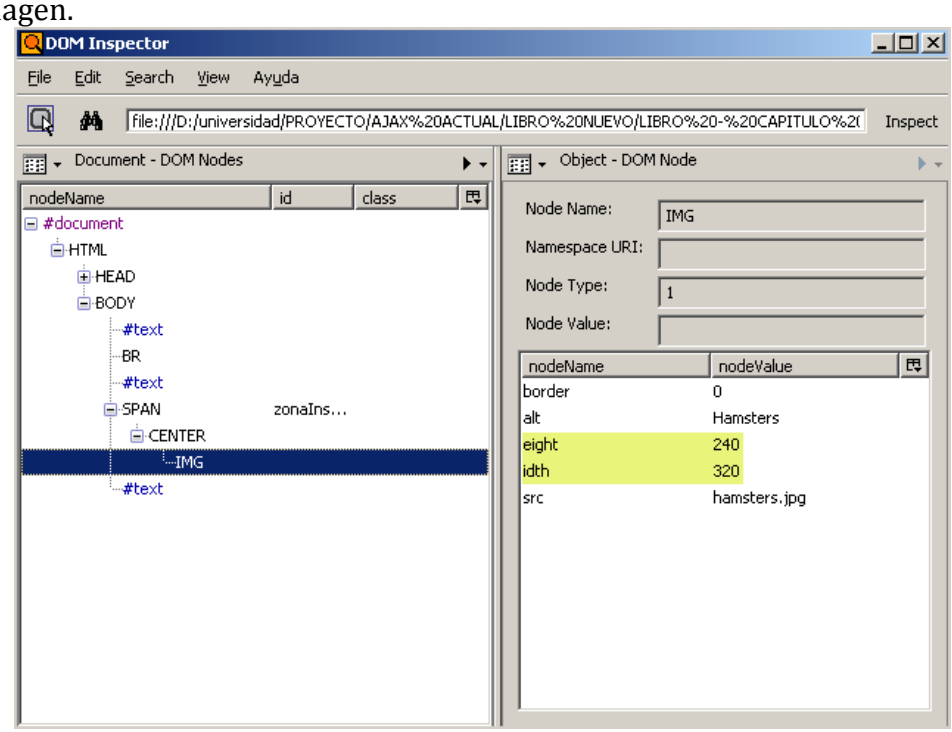


Ilustración 18 Ejemplo de uso de la utilidad DOM inspector.

Hemos insertado "border", "alt", "eight", "idth" y "src", vemos que "eight" y "idth" tienen los valores 240 y 320, lástima que todavía no existan como propiedades que interpretan los navegadores, las hemos escrito mal, corrijámoslo y problema solucionado.

2.5 Venkman(Depurador de Javascript)

Venkman es un depurador convencional, lo utilizaremos para poner puntos de ruptura en el código, de forma que Mozilla Firefox parará su ejecución y no nos dejará continuar hasta que lo digamos en el depurador. Si nos paramos dentro de una función, podremos ver cómo cambian los valores de sus variables locales sobre la marcha si hacemos un "paso por paso".

Vamos al ejemplo, es una versión modificada de depuracion1.html, algo mas compleja y que parece no tener sentido, nos valdrá para aprender algunas cosas.

depuracion3.html

```
<html>
<head><title>depuración</title></head>
<script language="JavaScript" type="text/javascript">
  function hagoalgo()
  {
    var nombre = "Juan" ;
    agoalgo2() ;
  }
  function hagoalgo2()
  {
    hagoalgoreal();
  }
  function hagoalgoreal()
```

```
  {
    longitud = nombre.length;
    alert("La longitud del nombre es : " + longitud);
  }
</script>
<body>
PÁGINA PARA HACER PRUEBAS DE DEPURACIÓN SIMPLES
<br>
<a href="javascript:hagoalgo();">Llamo Función hagoalgo</a>
<a href="javascript:hagoalgo2();">Llamo Función hagoalgo2</a>
<a href="javascript:hagoalgoreal();">Llamo Función hagoalgoreal</a>
</body>
</html>
```

Lo primero que vamos a hacer después de abrir la página es abrir Venkman (el depurador de JavaScript), la cosa debería quedar como en la figura 2.7 .

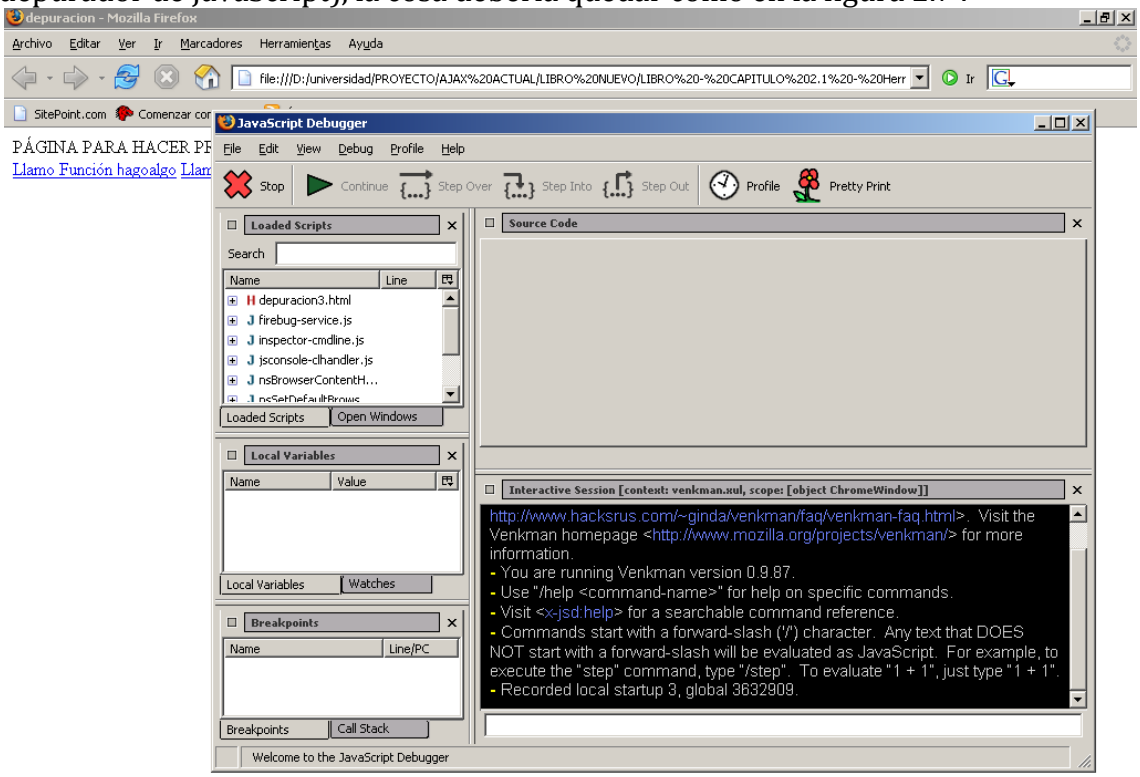


Ilustración 19 Pantalla de inicio de Venkman con el fichero depuracion3.html cargado en Firefox.

Solución de problemas (windows): Si alguna vez intentas abrir alguna herramienta de Mozilla Firefox y ésta no abre es porque de alguna manera se ha quedado "colgada" la última vez que la iniciaste. Sale del navegador y pulsa CONTROL+ALT+SUP, busca la pestaña de procesos y termina el proceso llamado "firefox" que sigue abierto incluso cuando tienes el navegador cerrado, esto ocurre normalmente con los depuradores si no se cierran correctamente.

Sigue estas instrucciones:
1. Abre la pestaña de H depuración3.html.
2. Acomódate la ventana para poder ver el código.
3. Pon los siguientes Puntos de ruptura pulsando con el botón izquierdo del ratón donde aparecen las B de (breakpoint) en la siguiente ilustración , verás que hay

unas barras horizontales, eso es porque es una zona seleccionable para un punto ruptura.

Te debería quedar todo como muestra la imagen.

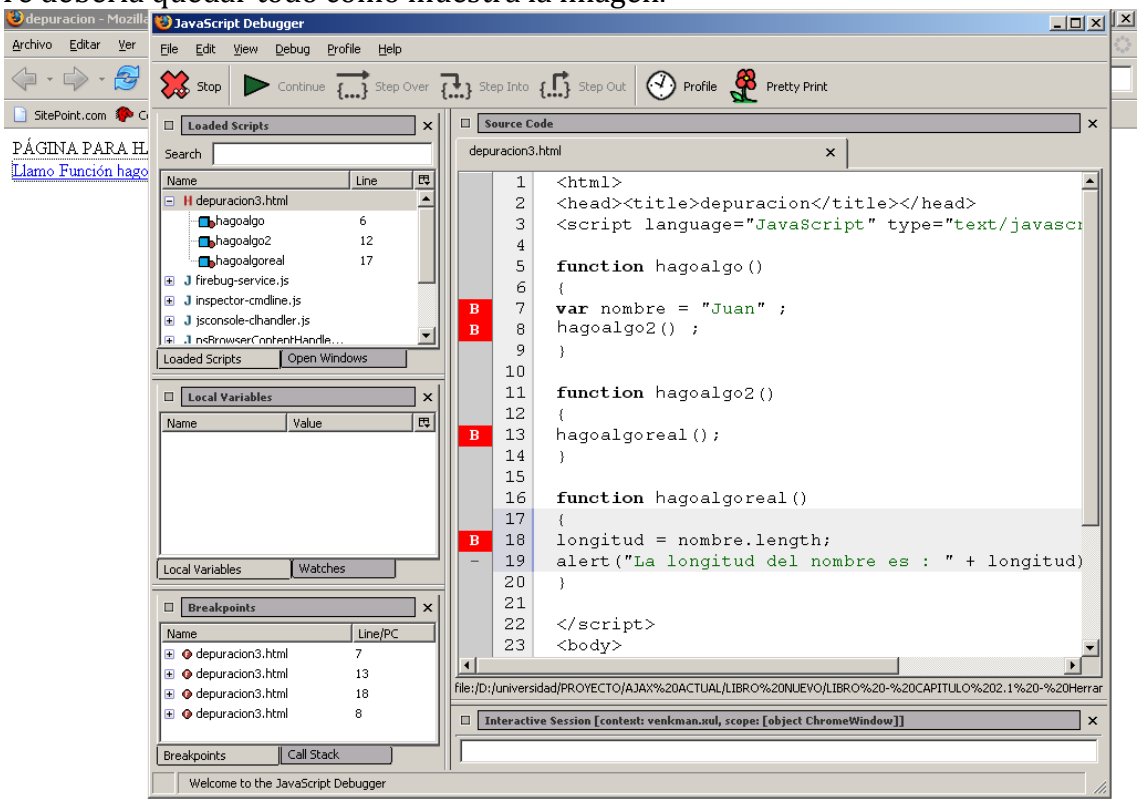


Ilustración 20 Venkman con el fichero depuracion3.html cargado, después de poner algunos breakpoints.

4. Ahora si se pulsa sobre el primer link, verás que cuando el código llegue al punto de ruptura se detendrá, como en la siguiente imagen.

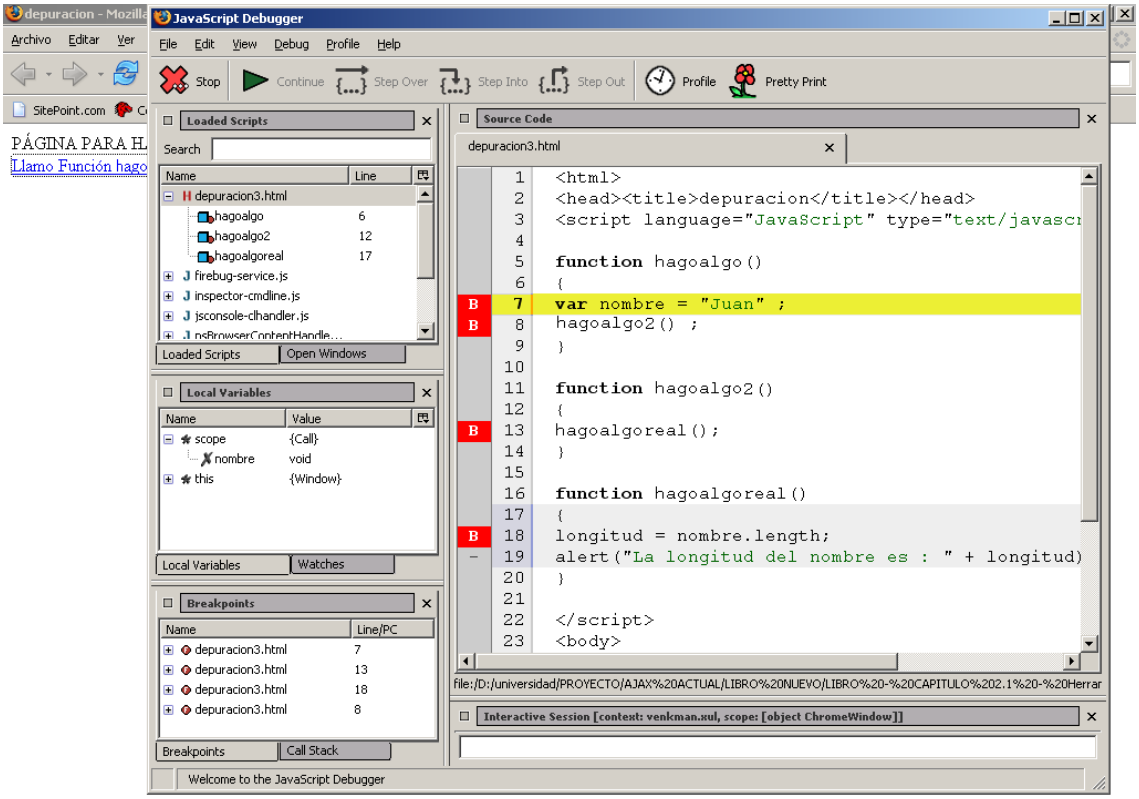


Ilustración 21 Venkman con el fichero depuracion3.html cargado, detenido por un punto de ruptura.

Si te fijas en la ventana de variables locales verás que la variable “nombre”, pero queremos hacer uso de esta variable en hagoalgoreal().

5. Vamos a ir a la ventana de watches y vamos a añadir una expresión con su nombre, esto será suficiente.
6. Damos un paso, con esto el estado debería ser el mostrado por la siguiente imagen.

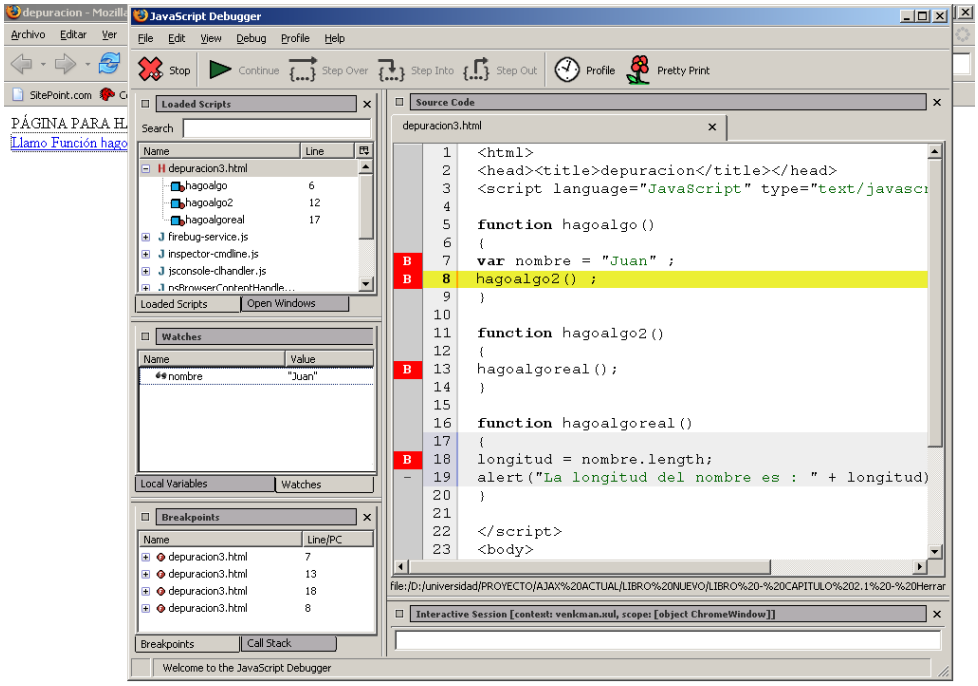


Ilustración 22 Vemos que la variable existe con su valor actual, tras el paso 6 de la explicación.

7. Damos varios pasos hasta llegar a la función hagoalgoreal().

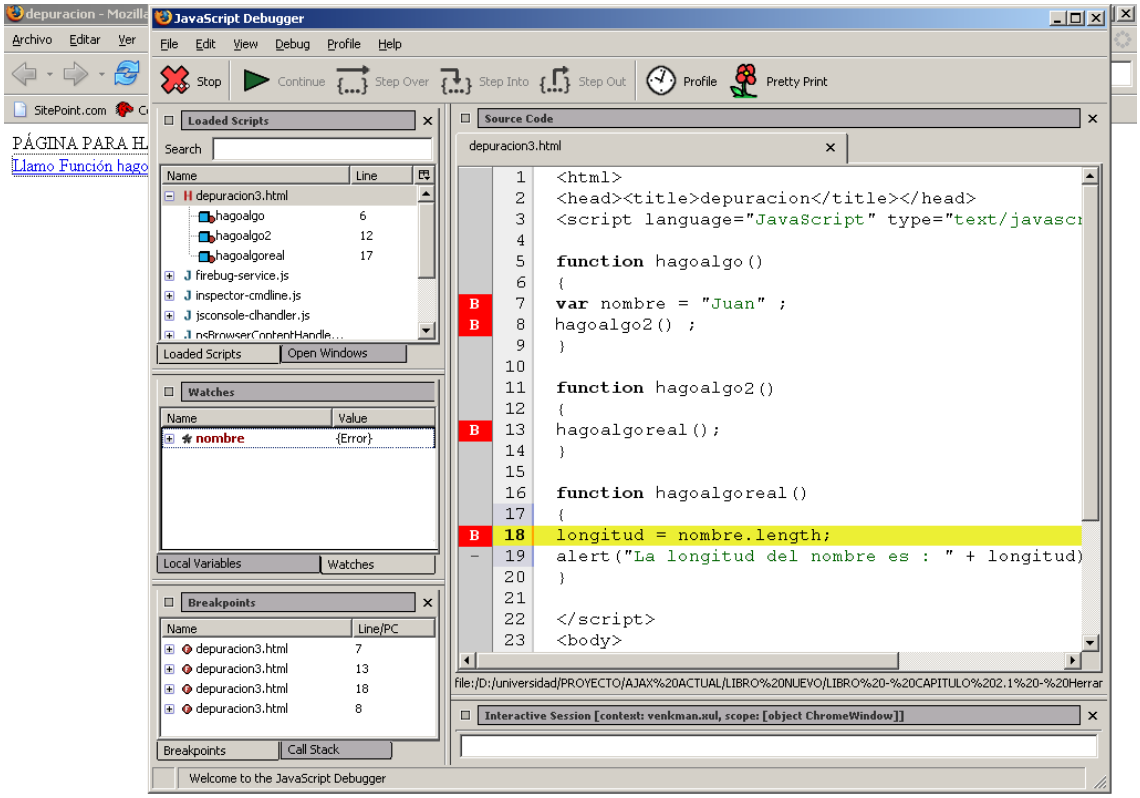


Ilustración 23 Estado tras el paso 7 de la explicación.

Vemos que ahora la variable nombre tiene el valor {Error}, esto es porque nada mas salirnos de la función, ya no la vemos, aunque haya sido llamada la función por una anterior que tiene la variable, no podemos acceder a ella, JavaScript no guarda la misma localidad que lenguajes como "C", es un fastidio, pero nos queda declararla como global o declarar una función dentro de otra como hemos hecho al hacer la ClasePeticionAjax.

Bueno, esto es básicamente lo que podemos hacer con el depurador que no es poco, nos permite examinar el código paso por paso viendo el valor de las variables que queramos.

2.6 FireBug (Todo lo anterior en uno).

Este es tal vez el depurador más nuevo y potente que hay, nos va a dar toda la información que nos dan los otros programas pero de forma mas rápida cuando nos acostumbremos.

Vamos a volver a utilizar el archivo depuracion3 para el ejemplo de su uso.

Lo abrimos en Mozilla Firefox y pulsamos sobre el icono de la esquina inferior derecha como muestra la siguiente imagen, de esta manera iniciamos FireBug.

Ilustración 24 Botón de inicio de FireBug.

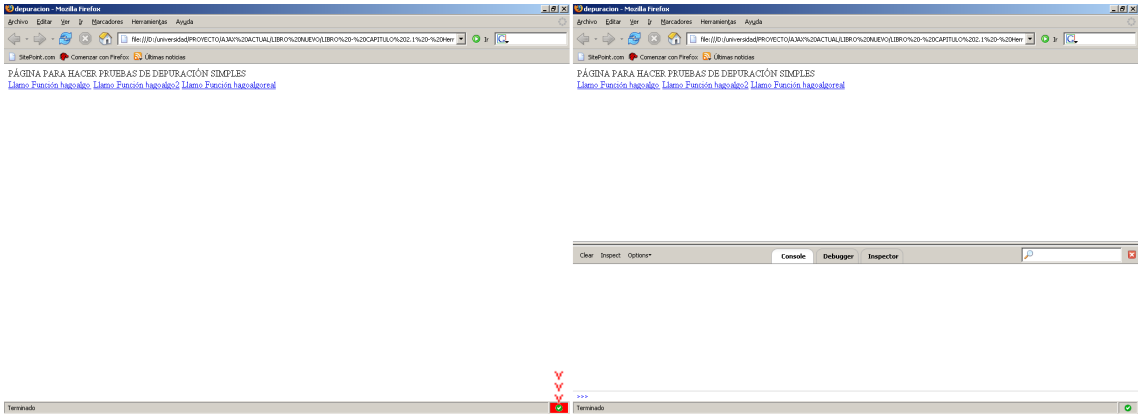
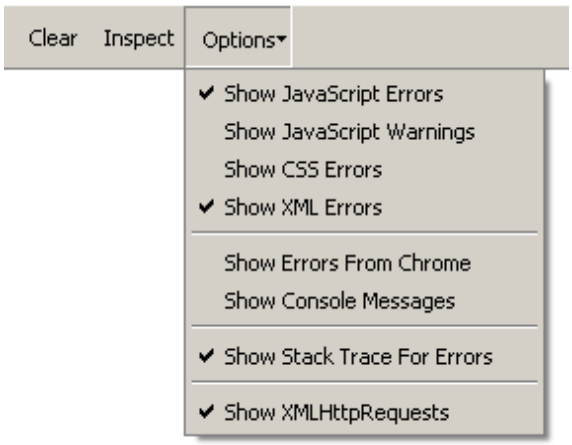


Ilustración 25 Ventana de FireBug abierta.

Tenemos 3 pestañas, veamos su funcionamiento una por una.

2.6.1 Pestaña de consola.

La consola nos dará toda la información que nos daba la consola de Javascript normal sumándole información extra que podemos mostrar si queremos.



Entre esta información extra, están los errores de las hojas de estilo CSS, los errores XML y dos que nos ayudarán mucho, mostrar la pila de llamadas que llegó hasta un error y mostrar las llamadas XMLHttpRequest, estos dos nos serán muy útiles. El botón "clear" simplemente borra toda la información acumulada en la consola y el "inspect" nos llevará a la pestaña del inspector.

Ilustración 26 Posibilidades de la consola de FireBug.

Hasta aquí una consola de errores convencional pero que nos muestra algo más de información, vamos a prestar atención a dos cosas, la pila de llamadas que nos muestra el camino hasta el error y los mensajes de error que podemos configurar nosotros mismos.

Para ver lo primero no tenemos más que tener seleccionado en opciones "Show Stack Trace For Errors" y pulsar sobre cualquier botón de nuestro ejemplo, yo voy a pulsar los 3 seguidos para que veamos bien como actúa en la siguiente ilustración.

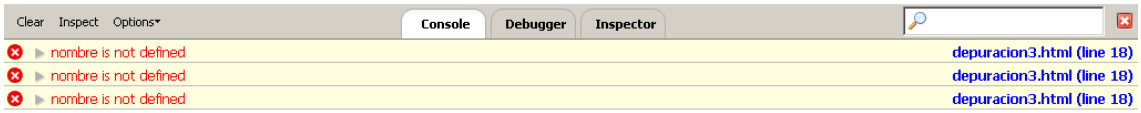


Ilustración 27 Errores mostrados por la consola de FireBug.

Vemos que las señales de error tienen una flecha al lado, mirando al error, dicen que no está definida donde se ha utilizado la variable "nombre".

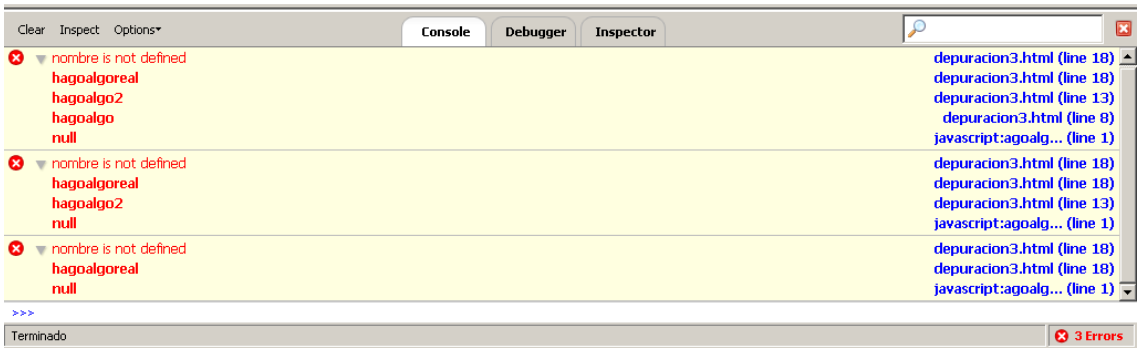


Ilustración 28 Pila de llamadas de los errores mostrados en la consola de FireBug.

Si pulsamos sobre la primera flecha se nos muestra la pila de llamadas, si llamamos a la primera función hagoalgo(), ésta llama a las 2 siguientes y termina en hagoalgoreal(), si seguimos la pila llegamos a la primera función, hagoalgo(), igual para el resto, de esta forma podemos seguir la secuencia de acontecimientos que desembocó en error.

Sobre los mensajes de error que podemos configurar nosotros mismos, estos se añaden al código y los interpreta FireBug, son los siguientes.

depuracion32.html con los mensajes que se pueden personalizar en amarillo.

```
<html>
<head><title>depuracion</title></head>
<script language="JavaScript" type="text/javascript">
   function consola()
   {
      var a = 10;
      var b = "hola";
      var c = [10, 20, 30, 40] ;
      console.log(a);  //Podemos poner simplemente una o varias variables.
      console.log(a,b);
      console.log("El valor de a es %d y el de b es %s",a,b); //Podemos usar la sintaxis del printf de c para
      los mensajes.
      console.info(c);
      console.warn(b);
      console.error("Prueba de error, el valor de b es %s.",b);
   }
   function hagoalgo()
   {
      var nombre = "Juan" ;
      hagoalgo2() ;
   }
   function hagoalgo2()
   {
      hagoalgoreal();
   }
   function hagoalgoreal()
   {
      longitud = nombre.length;
      alert("La longitud del nombre es : " + longitud);
   }
</script>
<body>
PÁGINA PARA HACER PRUEBAS DE DEPURACIÓN SIMPLES
<br>
<a href="javascript:hagoalgo();">Llamo Función hagoalgo</a>
<a href="javascript:hagoalgo2();">Llamo Función hagoalgo2</a>
<a href="javascript:hagoalgoreal();">Llamo Función hagoalgoreal</a>
<a href="javascript:consola();">Llamo Función consola</a>
</body>
</html>
```

Si ahora llamamos a la función “consola”, obtenemos el resultado mostrado en la siguiente ilustración.

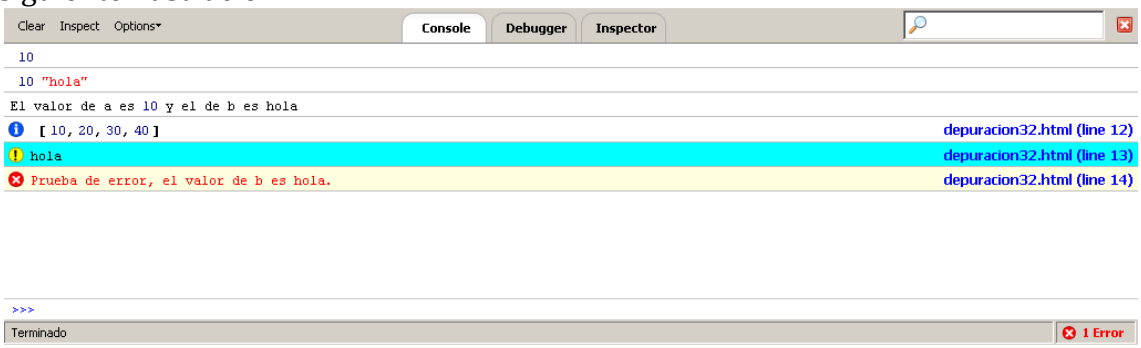


Ilustración 29 Los diferentes mensajes de error de FireBug, mostrados del archivo depuracion32.html.

- Los mensajes console.log() no muestran ningún cartel de atención especial.

- Los mensajes console.info() muestran un cartel de información.
- Los mensajes console.warn() muestran un cartel de alerta.
- Los mensajes console.error() muestran un cartel de error, pero el programa no se rompe.

El programa Javascript se romperá como haría normalmente si metemos en algún mensaje de error algo no válido como una variable que no existe.

2.6.2 Pestaña de debugger (depurador).

Funciona exactamente igual que Venkman, ponemos puntos de ruptura y cuando llegue el programa a este punto se parará, para seguir pulsamos sobre los botones, podemos ir paso a paso y se nos muestran las variables locales de la función en la parte derecha, voy a sumarle al ejemplo una variable global para que veamos como podrían verla desde dentro de la función.

Solución de problemas (windows): Si alguna vez te sale un mensaje de alarma en la consola cuando pones un punto de ruptura dentro del navegador, sal del navegador y pulsa CONTROL+ALT+SUP, vete a la pestaña de procesos y termina el proceso de Firefox que sigue abierto incluso cuando tienes el navegador cerrado, esto ocurre normalmente con los depuradores si no se cierran correctamente, además algunas veces tenemos que repetir varias veces el proceso en la pagina Web para que "coja" el punto de ruptura FireBug.

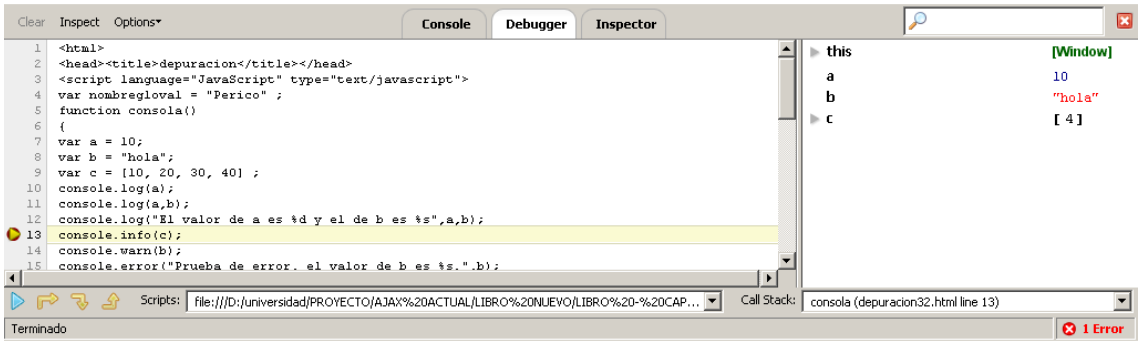


Ilustración 30 Pestaña de debugger de FireBug con un punto de ruptura.

No hemos salido de la pantalla navegador y estamos pudiendo probarlo todo abriendo el depurador, en esto le gana a Venkman, FireBug es mucho más cómodo.

Si queremos ver una variable global tenemos que abrir la pestaña de "window" en la parte derecha y buscar su valor, realmente estamos mirando entre el DOM.

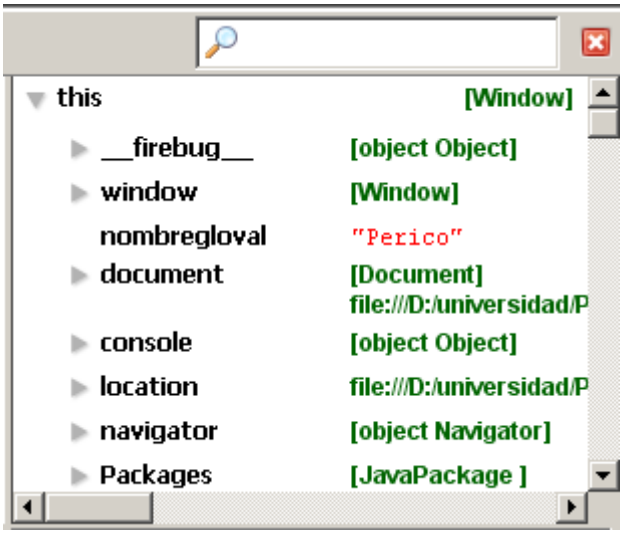


Ilustración 31 Mostrando el árbol de objetos JavaScript del DOM con FireBug.

La variable global se llamaba nombregloval y su valor es "Perico", encontrarla no ha sido tan cómodo como los watches de Venkman.

2.6.3 Pestaña del inspector.

Su uso en general es muy sencillo, tiene 5 sub. pestañas, nos centraremos en la pestaña Source. Si conocemos hojas de estilo y JavaScript utilizar el resto será muy sencillo.

Nos muestra el mismo código html que hemos escrito, pero nos deja desplegar las etiquetas y nos recuadra en el navegador la zona de la página que forma parte del código que se estamos señalando con el puntero del ratón.

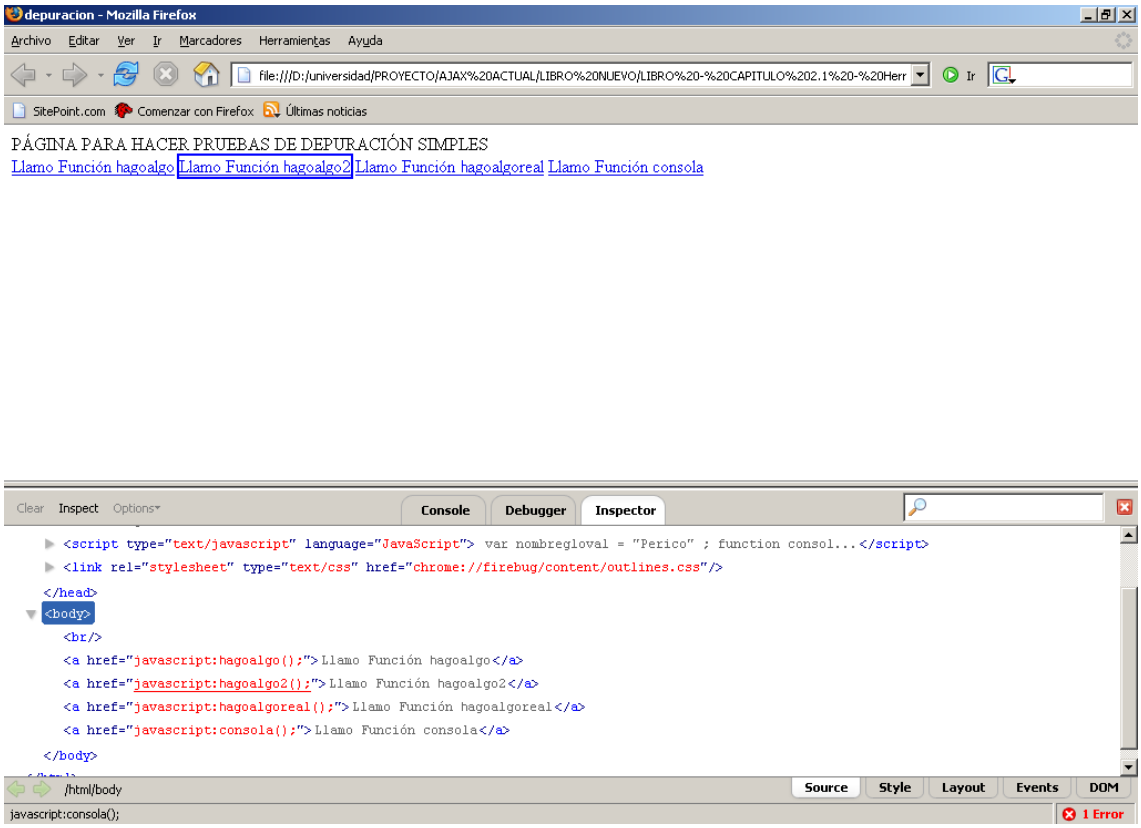


Ilustración 32 Inspector de código web de FireBug.

Como se ve por el subrayado estaba señalando el hiperenlace a la segunda función y en el navegador se recuadra, es muy sencillo de utilizar, lo malo es que nos muestra la página que cargamos al comenzar y si insertamos mucho código HTML dinámicamente nos podemos encontrar muchas veces con que no es coherente con lo que se está mostrando en el navegador, con esto terminamos el vistazo a FireBug.

Capítulo 3: Técnicas de petición de información

3.1 Descripción del capítulo.

Por ahora se ha visto todo lo relacionado al objeto XMLHttpRequest que es el corazón de AJAX y terminamos el capítulo anterior construyendo una librería que nos dejará utilizarlo de forma mas cómoda, pero que todavía no nos abstrae totalmente del uso del objeto XMLHttpRequest.

Lo realmente interesante de este objeto es utilizarlo para traer a nuestra página los componentes que se utilizan en las páginas Web actuales, HTML, JavaScript, imágenes, entre otros, pero estos son los básicos y después mejorar la librería para abstraernos del famoso objeto y tener una sencilla función que lo haga todo por nosotros.

Mientras que se trabaja con JavaScript es muy fácil que cometamos algún error, por lo que se han visto primero las herramientas de depuración.

En este capítulo vamos a ver:

3.2 Insertar Código HTML (Y con ello, lo que contenga el código).
3.3 Insertar Una imagen usando el DOM no como código HTML.
3.4 Insertar JavaScript.
3.5 El DOM y JavaScript.
3.6 Diferencia entre insertar código HTML directamente o crearlo con el DOM.
3.7 Encapsular el objeto XMLHttpRequest facilitando su uso.
3.8 Manejo de Errores en la librería.
3.9 Dar soporte al usuario mientras realizamos las peticiones.

Todo con sus ejemplos funcionando como siempre para que el lector pueda jugar con ellos y comprender bien su funcionamiento que es lo que se pretende.

3.2 Insertar código HTML.

Esto es lo más fácil, además nos ayudará a comprender algunas cosas de forma sencilla.

Vamos a seguir utilizando la librería que construimos en el apartado anterior, primero vamos a ver el ejemplo y luego se explicará, se van a resaltar las partes más importantes para explicarlas posteriormente.

insertarHTML.html

```
<html>
<head>
<title>Insertar HTML</title>
<script language="JavaScript"
src="lib/ConstructorXMLHttpRequest.js"></script>
<script language="JavaScript" src="lib/ClasePeticionAjax.js"></script>
<script language="JavaScript" type="text/javascript">
  var PeticionAjax01 = new objetoAjax(); //Definimos un nuevo
  objetoAjax.
  PeticionAjax01.completado = objetoRequestCompletado01; /*Función
  completado del objetoAjax redefinida. */
  function objetoRequestCompletado01(estado, estadoTexto,
  respuestaTexto, respuestaXML)
  {
    document.getElementById('ParteHTML').innerHTML = respuestaTexto;
    //Solo nos interesa la respuesta como texto
  }
</script>
</head>
<body>
<span id="ParteHTML"> //Principio del código que se va a sustituir
  <center>
  <button onclick="PeticionAjax01.coger('pag01.html')">Coge la
  pag01</button>
  </center>
</span> //Final del código que se va a sustituir
</body>
</html>
```

pag01.html

```
<center><b> Os presento al hamster guerrero </b> </center>
<center><img src="img/guerrero.jpg"><center>
```

img/guerrero.jpg

Ilustración 33 Imagen divertida de un hámster para amenizar el ejemplo.

Cargar en formato texto:

```
document.getElementById('ParteHTML').innerHTML = respuestaTexto;
//Solo nos interesa la respuesta como texto
```

Con esto busco que nos demos cuenta de que estoy introduciendo texto dentro del código HTML, pero da la casualidad de que ese texto, es mas código HTML y cuando el navegador lo interprete, lo interpretará como HTML que es, esto es porque lo está insertando en "innerHTML".

La parte del documento que sustituimos:

```
<center>
<button onclick="PeticionAjax01.coger('pag01.html')">Coge la
pag01</button>
</center>
```

Hasta ahora no había mencionado a la etiqueta que se lleva utilizando desde el primer ejemplo, ésta define una sección dentro del documento, en el caso que nos ocupa la sección comienza antes del botón y termina después de éste, además esa es la sección de código que vamos a sustituir, con lo que el botón desaparecerá de la página. Esto es muy útil, ya que podemos definir la página como una tabla y está en secciones, de forma que podemos ir cambiando secciones con un menú sin cambiar de página.

Tener claro donde esta nuestra página:

pag01.html

```
<center><b> Os presento al hamster guerrero </b> </center>
<center><img src="img/guerrero.jpg"><center>
```

Estamos en la página insertarHTML.html, por lo tanto no es necesario volver a insertar las cabeceras, <html>, <head>,<body>, etc.

img/guerrero.jpg

El navegador actúa de manera que el código nuevo formaba parte de "la página original", con esto quiero decir que si insertamos algo que contenga una dirección (como una imagen), esa dirección debe ser relativa a la página original, no a la que hemos cargado.

Dicho todo lo anterior, veamos el resultado en la siguiente imagen:

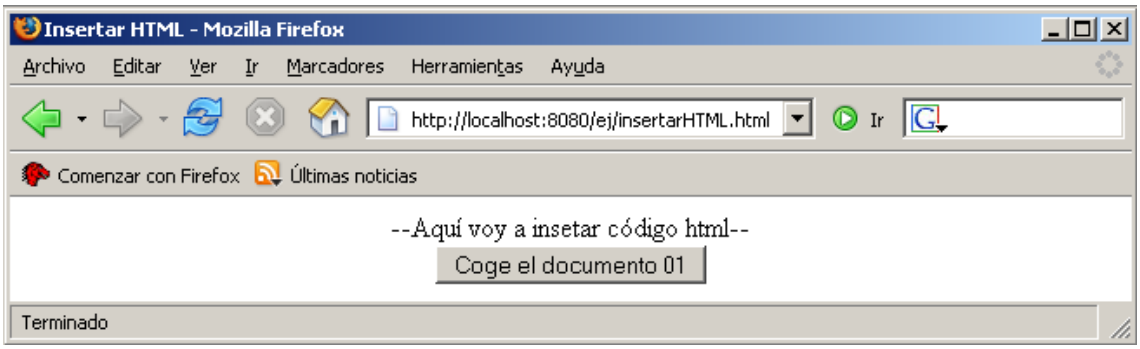


Ilustración 34 Primer ejemplo útil de AJAX.

Si pulsamos el botón la cosa queda como muestra la siguiente imagen:

Ilustración 35 Código HTML insertado correctamente.

Como vemos seguimos en la misma página ya que permanece el mensaje de "--Aquí voy a insertar código HTML--", además de que el texto ha sido interpretado como código HTML.

Pues esto es AJAX, una técnica que nos permite cargar en una misma página diferentes elementos, bajo demanda.

3.3 Insertar imágenes usando el DOM.

Aunque podemos insertar imágenes como HTML, también podemos crear la etiqueta utilizando el API del DOM, pero algo mas curioso es lo siguiente, debido a las necesidades de nuestra aplicación podemos querer que, aunque no esté la imagen o las imágenes cargadas, aparezca el hueco donde estarán en un futuro, como en principio parece que no tiene utilidad voy a mostrar un ejemplo práctico real de una aplicación Web AJAX.

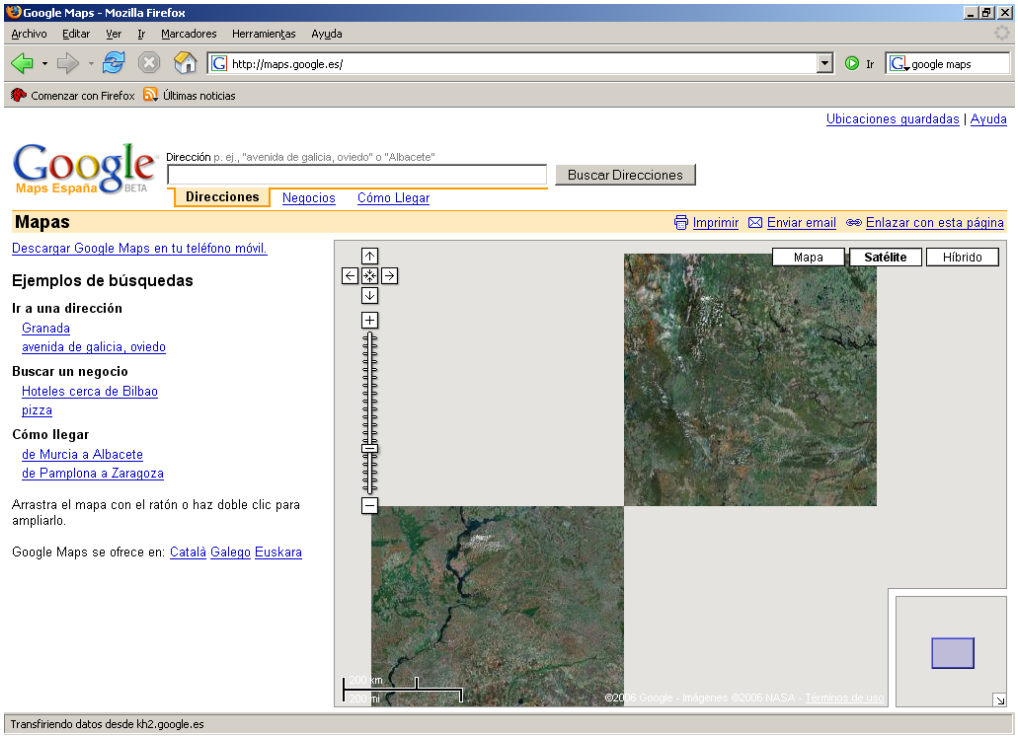


Ilustración 36 Google maps sin terminar de cargar imágenes.

Si nos fijamos en la barra de abajo del navegador, pone "transfiriendo datos....", mientras que estos datos que son las imágenes se transfieren, tienen su espacio ya guardado, de esta forma no se desfigura la representación del mapa. Por supuesto que la complejidad de google Maps no se haya en la inserción de las imágenes.

Veamos un ejemplo de cómo hacer esto; como en el apartado anterior se resaltará lo más importante para más tarde tratarlo con detalle.

insertarImag.html

```
<html>
<head>
<title>Insertar una imagen</title>
<script language="JavaScript"
src="lib/ConstructorXMLHttpRequest.js"></script>
<script language="JavaScript" src="lib/ClasePeticionAjax.js"></script>
<script language="JavaScript" type="text/javascript">
  var PeticionAjax01 = new objetoAjax(); //Definimos un nuevo
  objetoAjax.
  PeticionAjax01.completado = objetoRequestCompletado01; //Función
  completado del objetoAjax redefinida.
  function objetoRequestCompletado01(estado, estadoTexto,
  respuestaTexto, respuestaXML)
  {
    document.getElementById('Imagen01').src = respuestaTexto; //Solo nos
    interesa la respuesta como texto
  }
</script>

</head>
<body>
<center><button
onclick="PeticionAjax01.coger('DireccionImagen01.txt')">Coge la Imagen
01</button></center>
<br>
<!- Esta imagen no tiene el campo source -->
<center><img id="Imagen01" width="412" height="450" /></center>
</body>
</html>
```

DireccionImagen01.txt

```
img/gatito.jpg
```

Img/gatito.jpg

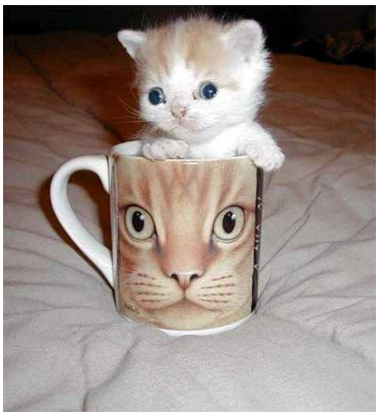


Ilustración 37 Una imagen graciosa para el ejemplo.

De este ejemplo podemos destacar 3 cosas:
La primera, es que le hemos otorgado a una imagen, un id, en vez de utilizar una etiqueta como hasta ahora y sustituir un campo del código, no solo eso sino que además le hemos puesto a la imagen el tamaño que tendrá, pero no el campo src donde se encuentra, de forma que en el cuadro que aparecerá vacío para una imagen podemos insertar la imagen que queramos.
Segundo, la dirección de la imagen se encuentra en un txt, podría estar en un fichero html, pero en contenido sería el mismo, las extensiones dan igual, estamos eligiendo un fichero de donde cargar texto, se ha puesto así para que quede más claro.

Y tercero, ¿Como le estamos diciendo que queremos cargar el campo src de una imagen?

```
document.getElementById('Imagen01').src
```

Si te fijas, en los ejemplos anteriores en vez de escribir ".src", escribíamos ".innerHTML".

Ahora toca explicar esto en más profundidad para los que no conocen el DOM (Document Object Model), estamos accediendo mediante JavaScript, primero a document que es nuestra pagina web, estamos eligiendo un elemento de la página con getElementById que es la imagen a la que le hemos dado un id, y por ultimo estamos accediendo al campo .src de la imagen01, que aunque no lo hemos escrito, existe, es nulo, pero existe, y estamos diciendo que es "=" a la respuesta en formato texto, con lo cual le acabamos de indicar dónde se encuentra la imagen, si se extrapola lo anterior esto se puede hacer con todo, la mente del lector puede empezar a maquinar.

En los ejemplos anteriores accedíamos al código HTML delimitado por la etiqueta , así que estábamos sustituyendo dentro de la propiedad que contiene el código HTML, por eso nuestro texto se interpretaba como código.

Con todo lo anterior veamos como queda el ejemplo en la siguiente imagen:

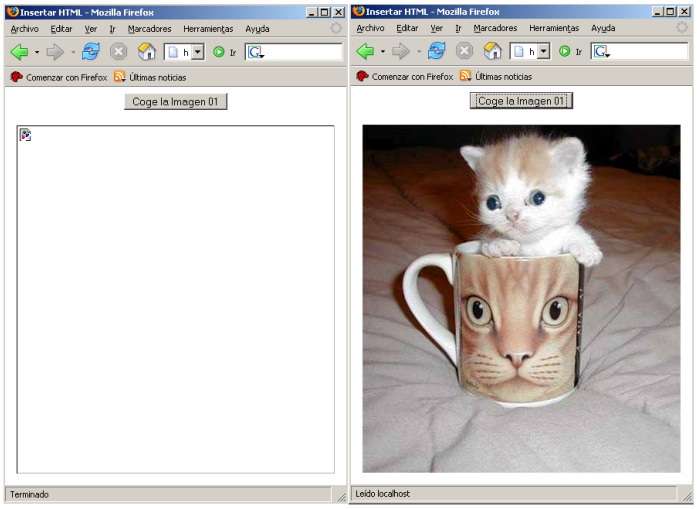


Ilustración 38 El antes y el después del ejemplo que nos ocupa.

Ahora ya sabemos cómo cambiar campos del DOM dinámicamente cargando su contenido mediante AJAX, en este caso, lo hemos utilizado para cargar el src de una imagen contenida en un txt, si se hubiera insertado directamente ya que conocíamos el camino sin hacer una petición al servidor para averiguarlo no sería AJAX, el uso del DOM tiene más posibilidades aunque como veremos mas tarde, su uso es muy tedioso para programadores inexpertos, sin sumarle las incompatibilidades entre navegadores, una vez aprendido y con práctica será una herramienta potentísima que será la culpable de poder hacer verdaderas virguerías.

3.4 Insertar código JavaScript.

Una cosa interesante, ya que puede ser necesaria a la hora de realizar ciertos programas, es la posibilidad de evaluar código JavaScript con la función eval(); lo bueno ahora, es que gracias a la técnica de programación AJAX, podemos recoger ese código de un archivo que podemos haber preparado antes o puede ser generado por el servidor dinámicamente y devuelto.

Un ejemplo sencillo de **llamar ejecutar código JavaScript con AJAX** sería:

insertarJavascript.html

```
<html>
<head>
<title>Insertar Javascript</title>
<script language="JavaScript"
src="lib/ConstructorXMLHttpRequest.js"></script>
<script language="JavaScript" src="lib/ClasePeticionAjax.js"></script>
<script language="JavaScript" type="text/javascript">
  var PeticionAjax01 = new objetoAjax(); //Definimos un nuevo
  objetoAjax.
  PeticionAjax01.completado = objetoRequestCompletado01; //Función
  completado del objetoAjax redefinida.
  function objetoRequestCompletado01(estado, estadoTexto,
  respuestaTexto, respuestaXML)
  {
    eval(respuestaTexto); //Solo nos interesa la respuesta como texto
    para lanzar el código JavaScript
  }
</script>
</head>
<body>
<center><button
onclick="PeticionAjax01.coger('CodigoJavascript01.txt')">Llama a una
Función</button></center>
</body>
</html>
```

CodigoJavascript01.txt

```
alert("Has llamado a un codigo javascript usando AJAX.");
```

El resultado se ve en la figura siguiente:

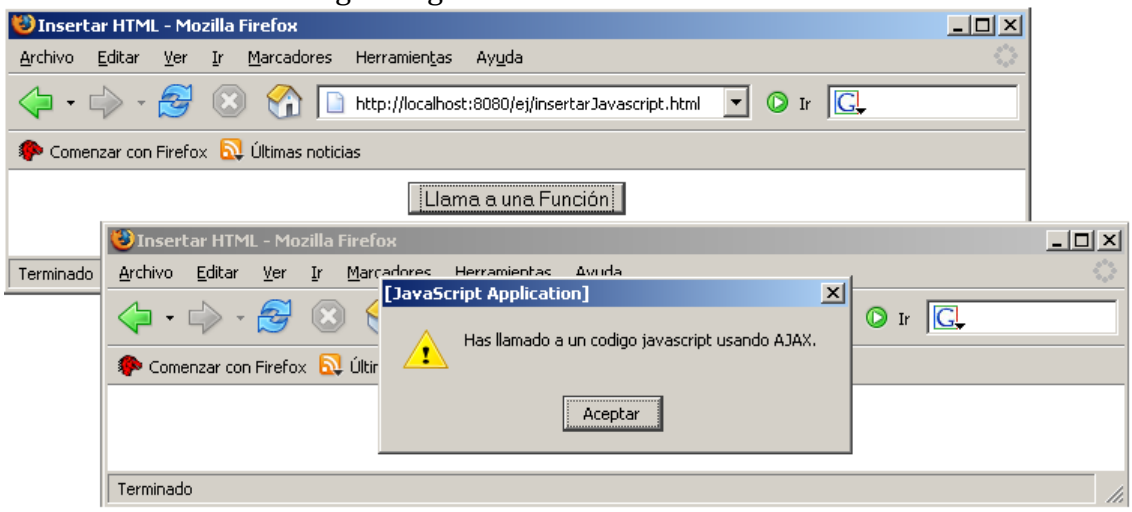


Ilustración 39 Código JavaScript devuelto por AJAX y evaluado.

Como vemos no es difícil y ya que podemos cargar código Javascript, también podemos cargar más peticiones AJAX, desde una petición AJAX, es decir, peticiones indirectas.

Un ejemplo de **petición indirecta** sería el siguiente:

insercionesIndirectas.html

```
<html>
<head>
<title>Insertar HTML</title>
<script language="JavaScript" src="lib/ConstructorXMLHttpRequest.js">
</script>
<script language="JavaScript" src="lib/ClasePeticionAjax.js">
</script>
<script language="JavaScript" type="text/javascript">
  var PeticionAjax01 = new objetoAjax(); //Definimos un nuevo
  objetoAjax.
  PeticionAjax01.completado = objetoRequestCompletado01; //Función
  completado del objetoAjax redefinida.
  function objetoRequestCompletado01(estado, estadoTexto,
  respuestaTexto, respuestaXML)
  {
    eval(respuestaTexto); //Solo nos interesa la respuesta como texto
    para lanzar el código JavaScript
  }
</script>
</head>
<body>

<center>
  <button>
  onclick="PeticionAjax01.coger('CodigoJavascriptCargaInd01.txt')">
  Llama a una Función
  </button>
</center>

<center>
  <span id="Lugar01"></span>
</center>

</body>
</html>
```

CodigoJavascriptCargaInd01.txt

```
//Hemos llamado a una peticionAjax
var PeticionAjax02 = new objetoAjax(); //Definimos un nuevo
objetoAjax.
PeticionAjax02.completado = objetoRequestCompletado02; //Función
completado del objetoAjax redefinida.

function objetoRequestCompletado02(estado, estadoTexto,
respuestaTexto, respuestaXML)
{
document.getElementById('Lugar01').innerHTML = respuestaTexto;
//Insertamos en el lugar01
}

PeticionAjax02.coger('pag01.html') //Cogemos el código HTML  con la
PeticionAjax02
```

Vamos a cargar a la pag01.html que hemos utilizado en un ejemplo anterior, aun así aquí está el código para que no haya confusión:

pag01.html

```
<center><b> Os presento al hamster guerrero </b> </center>
<center><img src="img/guerrero.jpg"><center>
```

Img/guerrero.jpg

Ilustración 40 El hámster guerrero contraataca.

Para hacer algo como esto, este ejemplo sería demasiado costoso, pero para retratar cómo usar una petición indirecta lo mejor es algo simple, hemos llamado a la primera petición AJAX, que ha ejecutado un código JavaScript contenido dentro de un fichero de texto, este contenía una llamada AJAX que cargaba un pedazo de código HTML dentro de un pedazo contenido por una etiqueta . Como el lector puede ver, no estamos más que dándole una vuelta más a todo lo que hemos empleado anteriormente, viendo las posibilidades de utilización; por supuesto a la hora de utilizarlo realmente deberemos elegir la más adecuada a la situación, las peticiones indirectas son buenas para que una petición haga generar al servidor la segunda petición de forma dinámica si estamos buscando una información que no se sabe dónde está y el servidor sí.

El resultado del ejemplo es este:

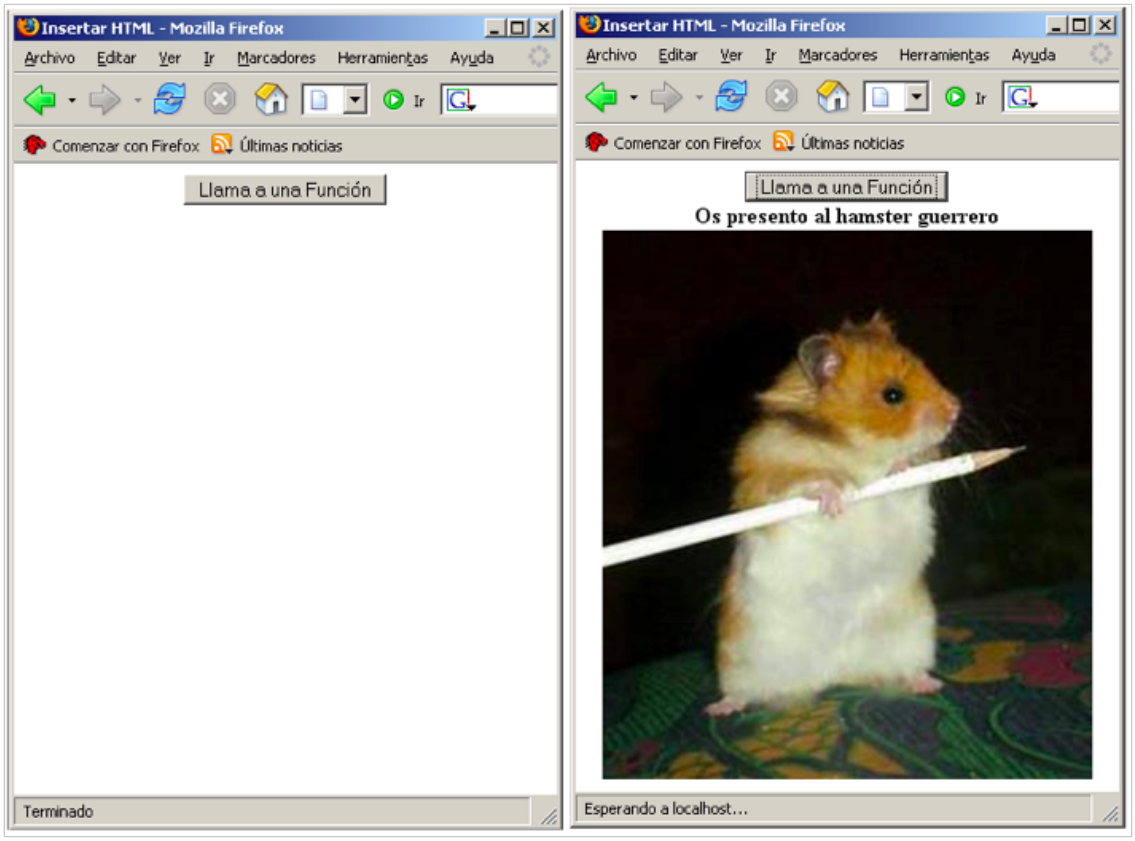


Ilustración 41 Código html cargado realizando una petición indirecta.

Con esto terminamos las técnicas básicas, ya hemos visto como cargar las cosas como texto, como introducirlas dentro de los objetos del DOM y como lanzar código JavaScript, todo utilizando AJAX.

3.5 DOM API.

El DOM esta íntimamente relacionado con JavaScript que aun por su nombre, no es un lenguaje orientado a objetos como java, es un lenguaje orientado a unos objetos, los del navegador, el DOM y los que creemos (diferentes de java), además carece de características importantes como la herencia, se basa en prototipos.

Ya que el lenguaje interactúa con los objetos del navegador, nos hacemos a la idea de que cuando descargamos una página HTML que contiene JavaScript, este código sigue estando dentro de la página ya que el lenguaje actúa en la parte del cliente, en cambio lenguajes como ASP, PHP o JSP generan lenguaje HTML desde el servidor y el usuario nunca llegará a ver su código en la página que descargue.

En el DOM el objeto principal es el objeto window:

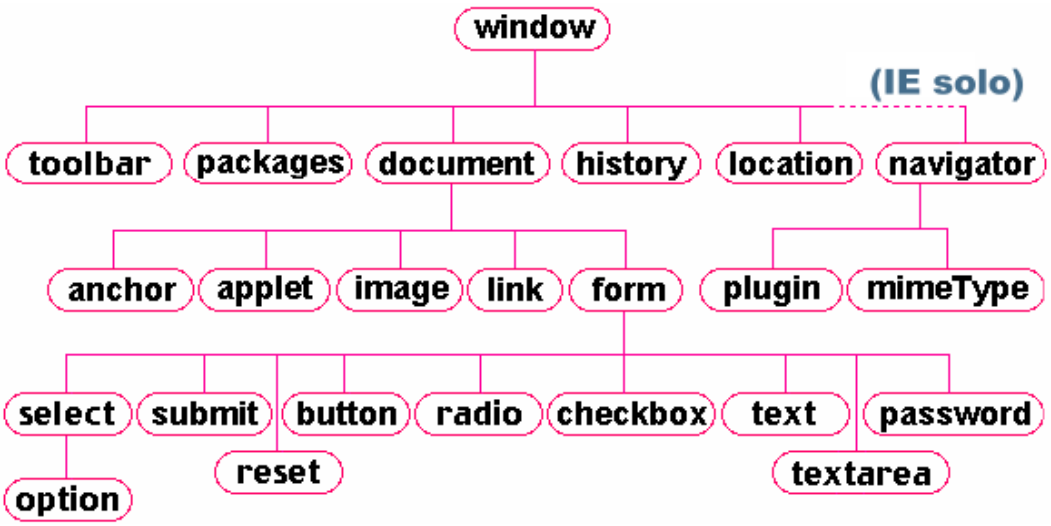


Ilustración 42 Jerarquía de objetos del navegador.

No es necesario (pero puedes hacerlo) implicar el objeto window en la llamada ya que está de forma implícita, como se ve en el siguiente ejemplo.

```
<html>
<body>
El dominio de esta página web es:
<script type="text/javascript">
document.write(document.domain)
</script>
</body>
</html>
```

Se han remarcado las etiquetas HTML que nos dejan insertar código JavaScript dentro de una página HTML convencional, el código JavaScript debe estar delimitado por estos tags de principio y fin, el resultado es el siguiente.

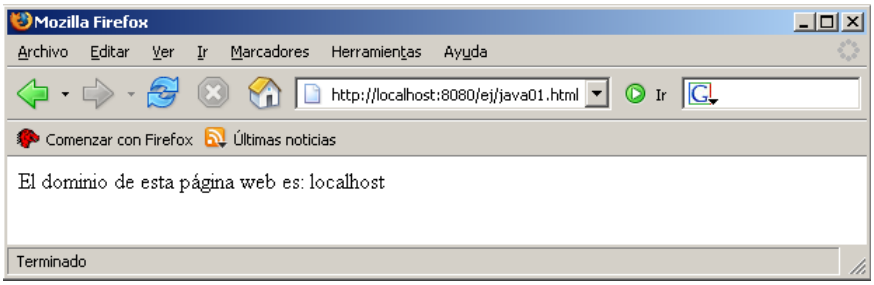


Ilustración 43 Ejemplo de uso de los objetos del navegador.

Internet Explorer extiende los objetos de navegador con algunos propios, el usarlos significaría la pérdida de compatibilidad con otros navegadores y obligaría a los usuarios a utilizar éste, no solo esto sino que ciertas cosas que se salen de las que podríamos llamar estándar se tratan de manera diferente dependiendo del navegador lo que obliga a crear código con bifurcaciones que primero detecte el navegador y dependiendo de este ejecute un pedazo de programa u otro sin dar errores.

Es normal que los diferentes toolkits que hay para programar aplicaciones Web orientadas a AJAX resuelvan estas incompatibilidades por el programador cuando

no queda remedio, en cambio en la mayoría de las ocasiones es posible encontrar un camino estándar que exista en todos los navegadores.

El árbol de objetos del navegador es un árbol general que contiene tanto métodos y propiedades del navegador como las que creemos nosotros mismos, si echamos un vistazo rápidamente con FireBug.

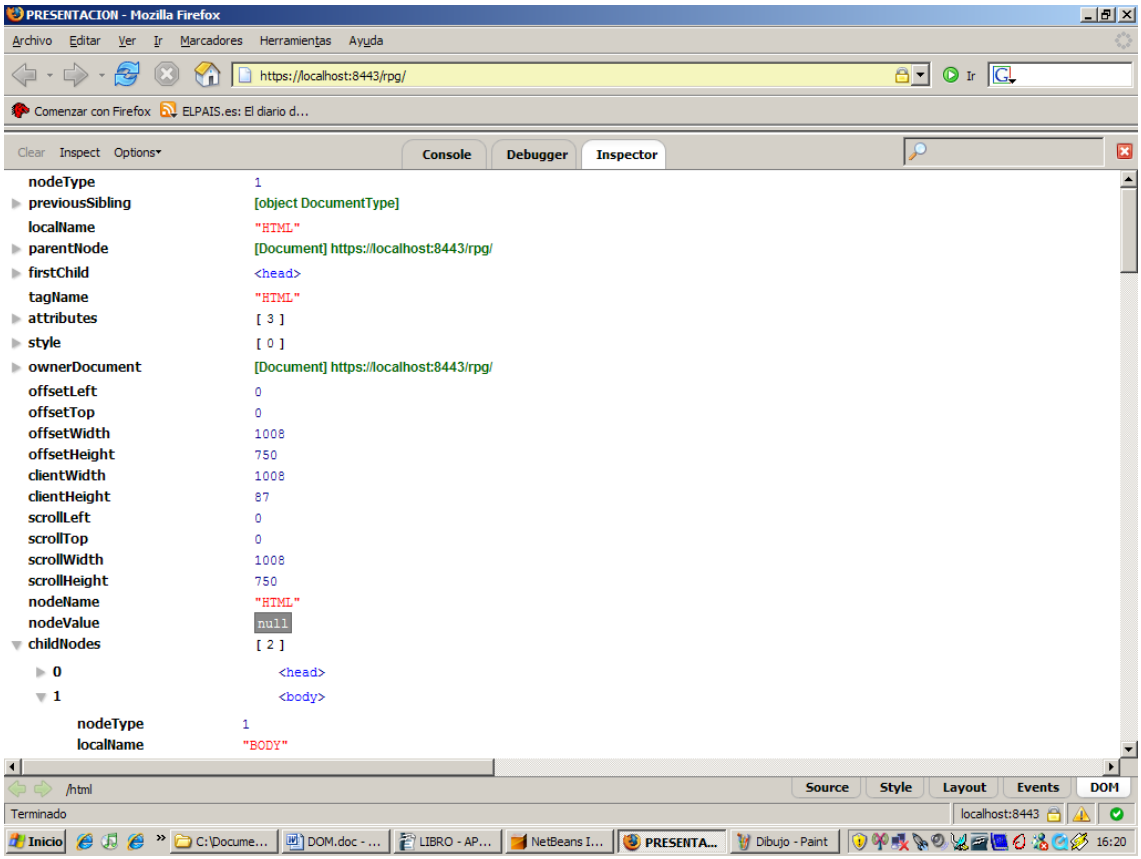


Ilustración 44 DOM de una página web en Mozilla Firefox 1.5 visto con FireBug.

Dentro del DOM cada etiqueta tiene una propiedad llamada innerHTML, hasta ahora modificando esta hemos modificado el documento, es decir cambiando el texto HTML cambia la página, en cambio también se puede modificar el documento cambiando las diferentes propiedades de cada nodo, esto es más complejo y laborioso pero tiene su recompensa cuando estamos creando entornos realmente atractivos, podemos ir creando las etiquetas, insertando texto y ellas y modificar sus propiedades programando en JavaScript.

3.6 DOM API e innerHTML enfrentados.

Hasta este momento hemos estado utilizando la propiedad InnerHTML de las etiquetas que delimitan zonas de la página y en ningún momento lo hemos discutido, vamos a ver otra posibilidad que aun existiendo, en principio y a no ser que surgiera algo que lo requiriera forzadamente su uso en ningún caso utilizaremos debido a que dispara la complejidad de la solución del problema.

Como dispara la complejidad, vamos a hacer algo tan sencillo como insertar una línea de texto plano, así comprenderemos rápidamente que no es una posibilidad

que debiéramos escoger por gusto, además obliga al estudio en mas profundidad del DOM.

La causa que hace que modificar el documento haciendo uso de innerHTML sea más fácil que modificar el DOM es que mientras que innerHTML modifica la estructura textual del documento, al usar el DOM modificamos la estructura lógica que se encuentra a un nivel más bajo.
Para comprender el siguiente código debemos entender que el DOM de una página Web es un árbol General, así pues un nodo puede tener muchos hijos, si sustituimos el nodo directamente y no ponemos sus hijos al nodo nuevo, habremos destrozado la página, debemos llevar cierto cuidado.

dom api.html

```
<html>
<head>
<title>Insertar con DOMAPI</title>
<script language="JavaScript"
src="lib/ConstructorXMLHttpRequest.js"></script>
<script language="JavaScript" src="lib/ClasePeticionAjax.js"></script>
<script language="JavaScript" type="text/javascript">
 var PeticionAjax01 = new objetoAjax(); //Definimos un nuevo
 objetoAjax.
 PeticionAjax01.completado = objetoRequestCompletado01; //Función
 completado del objetoAjax redefinida.
 function objetoRequestCompletado01(estado, estadoTexto,
 respuestaTexto, respuestaXML)
 {
   var viejaParte = document.getElementById('ParteHTML'); /*Recogemos
   el elemento que vamos a cambiar como

   normalmente./*
   var nuevaParte = document.createTextNode(respuestaTexto); /*Creamos
   un nuevo nodo de texto, con el texto de

   respuesta.*/
   if (viejaParte.childNodes[0]) //Vemos si el nodo tenía hijos.
   {
     viejaParte.replaceChild(nuevaParte, viejaParte.childNodes[0]);
     /*Cambiamos el nodo hijo del padre por lo nuevo */
   }
   else
   {
     viejaParte.appendChild(nuevaParte); //Le añadimos un hijo nuevo
     sino tiene
   }
 }
</script>

</head>
<body>

<button onclick="PeticionAjax01.coger('Texto.txt')">Coge el documento
01</button>
<span id="ParteHTML">
--Aquí voy a insertar Texto--
</span>

</body>
</html>
```

He señalado en azul claro el código necesario para utilizar el DOM directamente, como vemos es trabajar con las funciones de un árbol y recordemos que no estamos dando formato al texto, aunque ahora podríamos usar hojas de estilo CSS

como piden los documentos XHTML estrictos, la verdad es que debido a lo estrictos que son se termina poniendo la extensión html normal para que algún navegador no salte por cualquier error.

Ahora con InnerHTML:

innerhtml.html

```
<html>
<head>
<title>Insertar con DOMAPI</title>
<script language="JavaScript"
src="lib/ConstructorXMLHttpRequest.js"></script>
<script language="JavaScript" src="lib/ClasePeticionAjax.js"></script>
<script language="JavaScript" type="text/javascript">
  var PeticionAjax01 = new objetoAjax(); //Definimos un nuevo
  objetoAjax.
  PeticionAjax01.completado = objetoRequestCompletado01; //Función
  completado del objetoAjax redefinida.
  function objetoRequestCompletado01(estado, estadoTexto,
  respuestaTexto, respuestaXML)
  {
    document.getElementById('ParteHTML').innerHTML = respuestaTexto;
    //inserción innerHTML típica
  }
</script>

</head>
<body>

<button onclick="PeticionAjax01.coger('Texto.txt')">Coge el documento
01</button>
<span id="ParteHTML">
--Aquí voy a insertar Texto--
</span>

</body>
</html>
```

texto.txt

```
Una linea de texto.
```

Como vemos después de esto y todos los ejemplos anteriores, mientras insertemos algo como código html, la complejidad es la misma.

El resultado del ejemplo anterior en los 2 casos es el mostrado por la imagen siguiente

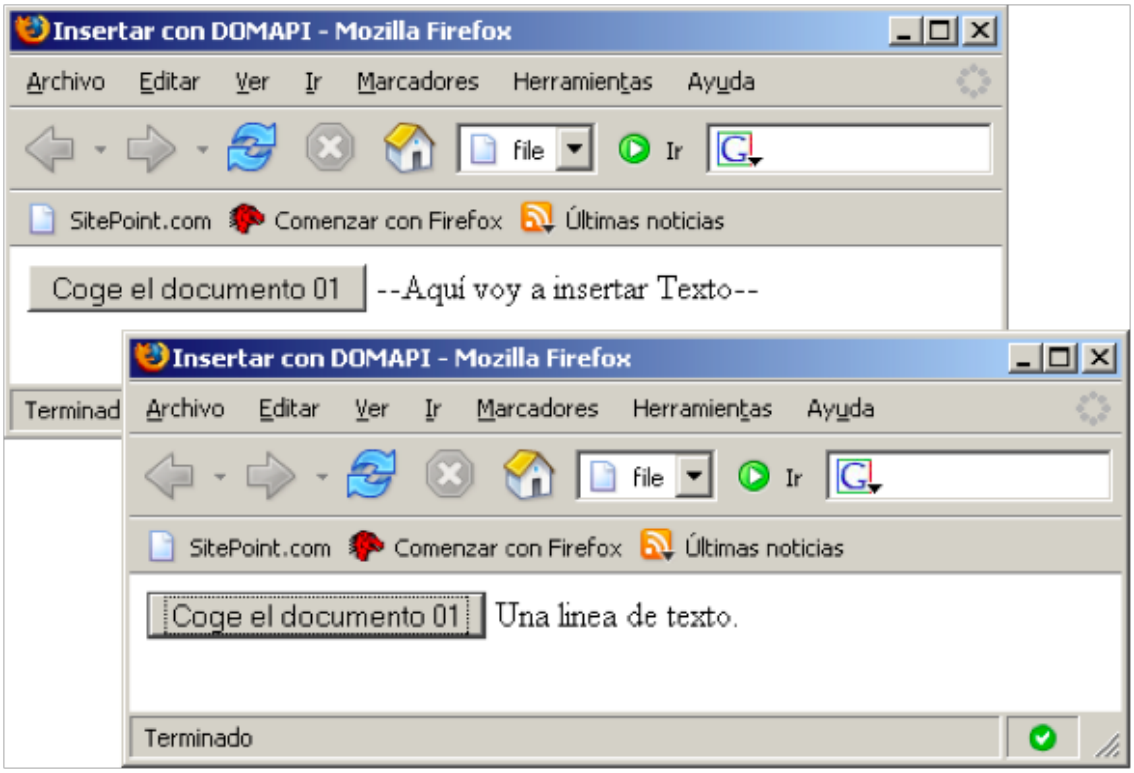


Ilustración 45 Ejemplo de inserción con innerHTML y DOM, todavía no usamos acentos al insertar código.

3.7 Encapsulación del objeto XMLHttpRequest.

Ya hemos visto algunas cosas que se pueden hacer con el objeto XMLHttpRequest que no son pocas, si le añadimos un programa en el lado del servidor que nos dé la información las posibilidades son muchísimas debido a que ya sabemos cómo pedir código en formato de texto, imágenes y JavaScript. Lo siguiente será poder hacerlo con una línea de código y olvidarnos de la programación a bajo nivel. Es lo que haremos, te puedes imaginar que estas a punto de leer un montón de código; tómatelo con calma, ejecútalo y mira como actúa, esta es la parte más fea de este tema.

3.7.1 Petición de código HTML o texto.

Como siempre, es mejor verlo con un ejemplo, voy a reutilizar archivos de los ejemplos anteriores ya que estamos familiarizados con ellos y se comprenderá más fácilmente.

insertarhtml.html

```
<html>
<head>
<title>Insertar HTML</title>
<script language="JavaScript" src="lib/ConstructorXMLHttpRequest.js">
</script>
<script language="JavaScript" src="lib/ClasePeticionAjax.js">
</script>
<script language="JavaScript" src="lib/ClasePeticionAjaxHtml.js">
</script>
<script language="JavaScript" type="text/javascript">
  var PeticionAjaxHtml01 = new
  objetoAjaxHtml("pag01.html","ParteHTML"); /*Pedimos una página y
  decimos  en que campo colocarla. */
</script>
</head>
<body>
<center>--Aquí voy a insertar código html--</center>
<span id="ParteHTML">
  <center>
    <button onclick="PeticionAjaxHtml01.cogerHtml()">Coge una pag html
    </button>
  </center>
</span>
</body>
</html>
```

Y la clase que nos permite hacerlo es la siguiente.

ClasePeticionAjaxHtml.js

```
/* Objeto que crea el automáticamente el XMLHttpRequest, pide la
información que recoge como texto y la inserta en el sitio pedido. */

function objetoAjaxHtml(ruta,idDondeInsertar)
{
this.ruta = ruta; //Ruta que llega asta el archivo con su nombre y
extensión
this.id = idDondeInsertar; //El campo donde insertar
}

function cogerHtml()
{
      var idActual = this.id ; /*Dentro de las funciones el this. no
funcionara, así que creamos una variable con su contenido, como
anteriormente.*/
      this.completado = function(estado, estadoTexto, respuestaTexto,
respuestaXML)
      {
      document.getElementById(idActual).innerHTML = respuestaTexto;
      }
this.coger(this.ruta); /*Si alguien ha llamado a la función cogerHtml
es porque quiere lanzar la petición y nosotros lanzamos la petición
xmlhttprequest. */
}
//Esta nueva clase hereda como prototipo la ClasePeticionAjax
objetoAjaxHtml.prototype = new objetoAjax;
//Definimos las funciones nuevas pertenecientes al objeto Html en
particular.
objetoAjaxHtml.prototype.cogerHtml  =  cogerHtml;  //Le  añadimos  la
función cogerHtml.
```

Con todo esto cogeríamos la misma página del ejemplo del ratón y el resultado sería el mismo que anteriormente, cabe destacar el uso de prototipos, ésta clase es un objetoAjax especializado para el manejo de HTML.

3.7.2 Petición de la dirección de una imagen

Anteriormente hemos visto dos tipos de inserción con el innerHTML(tipo1) y con el DOM(tipo2), la clase que haga esta tarea nos dejará hacer los dos tipos de inserciones.

insertarImagen1.html

```
<html>
<head>
<title>Insertar Imagen</title>
<script language="JavaScript" src="lib/ConstructorXMLHttpRequest.js">
</script>
<script language="JavaScript" src="lib/ClasePeticionAjax.js">
</script>
<script language="JavaScript" src="lib/ClasePeticionAjaxImagen.js">
</script>
<script language="JavaScript" type="text/javascript">
//Definimos un nuevo objetoAjaxImag de tipo 1  y las propiedades de la
imagen
//De izquierda a derecha las propiedades son las siguientes
//El archivo que devuelve la dirección de la imagen.
//Donde insertar la imagen una vez obtenida
//Tipo de inserción 1=InnerHtml 2=DOM
//Alto, Ancho, borde y alt
var PeticionAjaxImag01 = new
objetoAjaxImagen("direccion.txt","ParteHTML",1,"391","350","0","guerre
ro");
</script>

</head>

<body>
<center>--Aquí voy a insetar código html--</center>
<span id="ParteHTML">
      <center>
            <button onclick="PeticionAjaxImag01.cogerImagen()">Coge
una imagen tipo 1</button>
      </center>
</span>
</body>
</html>
```

En cambio si vamos a insertar directamente en el DOM, suponemos que las características de la imagen ya están en la página y solo necesitamos que nos devuelvan su dirección.

insertarImagen2.html

```
<html>
<head>
<title>Insertar Imagen</title>
<script language="JavaScript" src="lib/ConstructorXMLHttpRequest.js">
</script>
<script language="JavaScript" src="lib/ClasePeticionAjax.js">
</script>
<script language="JavaScript" src="lib/ClasePeticionAjaxImagen.js">
</script>
<script language="JavaScript" type="text/javascript">
//Definimos un nuevo objetoAjaxImag de tipo 1  y las propiedades de la
imagen
//De izquierda a derecha las propiedades son las siguientes
//El archivo que devuelve la dirección de la imagen.
//Donde insertar la imagen una vez obtenida
//Tipo de inserción 1=InnerHtml 2=DOM
var PeticionAjaxImag01 = new
objetoAjaxImagen("direccion.txt","ParteImag",2);
</script>
```

```
</head>

<body>
<center>--Aquí voy a insertar código html--</center>
<span id="ParteHTML">
      <center>
            <button onclick="PeticionAjaxImag01.cogerImagen()">Coge
una imagen tipo 2</button>
      </center>

      <center>
            <img id="ParteImag" height="391" width="350" />
      <center>
</span>
</body>
</html>
```

direccion.txt

```
guerrero.jpg
```

La clase que nos permite operar de la forma anterior sería la siguiente.

ClasePeticionAjaxImagen.js

```
function objetoAjaxImagen(ruta,idDondeInsertar,tipoInsercion,
alto,ancho,borde,alter) {
this.ruta = ruta; //Ruta que llega asta el archivo que contiene la
dirección del a imagen.
this.id = idDondeInsertar; //El campo donde insertar
this.tipoInsercion = tipoInsercion; //Tipo insección 1=InnerHtml 2=DOM
//Propiedades de la imagen
this.alto = alto;
this.ancho = ancho;
this.borde = borde;
this.alternativa = alter;
}

function cogerImagen() {
/*Dentro de las funciones el this. no funcionara, así que creamos una
variable nueva con su contenido, como anteriormente.*/
var idActual = this.id;
var tipoInsercionActual = this.tipoInsercion;
var anchoActual = this.ancho;
var altoActual = this.alto;
var bordeActual = this.borde;
var alterActual = this.alter;

this.completado = function(estado, estadoTexto,
respuestaTexto, respuestaXML) {
var rutaImagen = respuestaTexto;
switch(tipoInsercionActual) { //Realizamos la inserción
case 1: /* Inserción Tipo 1, insertamos código XHTML según sus
especificaciones, es decir, con las comillas para los atributos y el
cierre al final.*/
document.getElementById(idActual).innerHTML =  "<center><img src= \""
+ rutaImagen + "\" width=\"" + anchoActual + "\" height=\"" +
altoActual + "\" border=\"" + bordeActual + "\" alt=\"" + alterActual
+ "\" /></center>" ;
break;

case 2: //Insercion Tipo 2, insertamos directamente en el DOM
document.getElementById(idActual).src = rutaImagen;
break;
}
```

```
}

this.coger(this.ruta); /* Cogemos la ruta que contiene la dirección de
la imagen, ¡¡NO ES LA IMAGEN, SI SUPIERAMOS DONDE ESTA PARA QUE
QUEREMOS AJAX!!  (Para los que no se hayan enterado todavía, si
sabemos dónde está la imagen, podemos cambiar su src
document.getElementById(idActual).src = rutaImagen;
directamente en Javascript sin necesidad de invocar al servidor
mediante el objeto XMLHttpRequest) */
}

//Esta nueva clase hereda como prototipo la ClasePeticionAjax y la
extiende.
objetoAjaxImagen.prototype = new objetoAjax;
//Funciones propias solo de esta clase.
objetoAjaxImagen.prototype.cogerImagen = cogerImagen; //La función de
coger propia de las imágenes.
```

El resultado de lo anterior sería mostrar la imagen del ratón, no añade nada nuevo, así que la omitimos también.

3.7.3 Petición de código JavaScript y lanzarlo.

Este es con diferencia la clase más sencilla, se ha dejado para el final para las mentes cansadas.

LanzarJavascript.html

```
<html>
<head>
<title>Insertar JavaScript</title>
<script language="JavaScript" src="lib/ConstructorXMLHttpRequest.js">
</script>
<script language="JavaScript" src="lib/ClasePeticionAjax.js">
</script>
<script language="JavaScript" src="lib/ClasePeticionAjaxJavascript.js">
</script>
<script language="JavaScript" type="text/javascript">
var PeticionAjaxjavascriptl01 = new
objetoAjaxJavascript("alertaejemplo.js"); /* Definimos un nuevo
objetoAjaxJavascript */
</script>

</head>
<body>
      <center>
            <button
onclick="PeticionAjaxjavascriptl01.cogerJavascript()">Lanza código
Javascript</button>
      </center>
</body>
</html>
```

ClasePeticionAjaxJavascript.js

```
function objetoAjaxJavascript(ruta)  //Solo necesitaremos la ruta.
{
this.Ruta = ruta; //Ruta, que además podemos redefinir durante el
programa.
}

function cogerJavascript()
{
      this.completado = function(estado, estadoTexto, respuestaTexto,
respuestaXML)
                  {
                  eval(respuestaTexto); //Lanzamos el código
```

```
JavaScript.
                    }
      this.coger(this.Ruta); //Si alguien lanza la petición nosotros
hacemos lo mismo.
}

//Esta nueva clase hereda como prototipo la ClasePeticionAjax
objetoAjaxJavascript.prototype = new objetoAjax;
//Prototipos propios de la clase objetoAjaxJavascript
objetoAjaxJavascript.prototype.cogerJavascript    =    cogerJavascript;
/*Añadimos la función de coger al objeto. */
```

alertaejemplo.js

```
alert("Has llamado a un código javascript usando AJAX.");
```

Con todo lo anterior ahora podemos aprovechar AJAX abstrayéndonos de tener que pensar en el objeto XMLHttpRequest lo que es un gran avance.

3.8 Manejo de errores.

Podemos preguntarnos un momento qué ocurre si hacemos una petición de información a una url inexistente dentro de un servidor que sí existe, puedes introducir una en tu navegador, el servidor te devolverá el código de error 404, página no encontrada.

Tal y como está hecha ClasePeticionAjax.js ahora mismo, aunque la url no existiese se entraría en estado 4 y ocurriría lo siguiente.

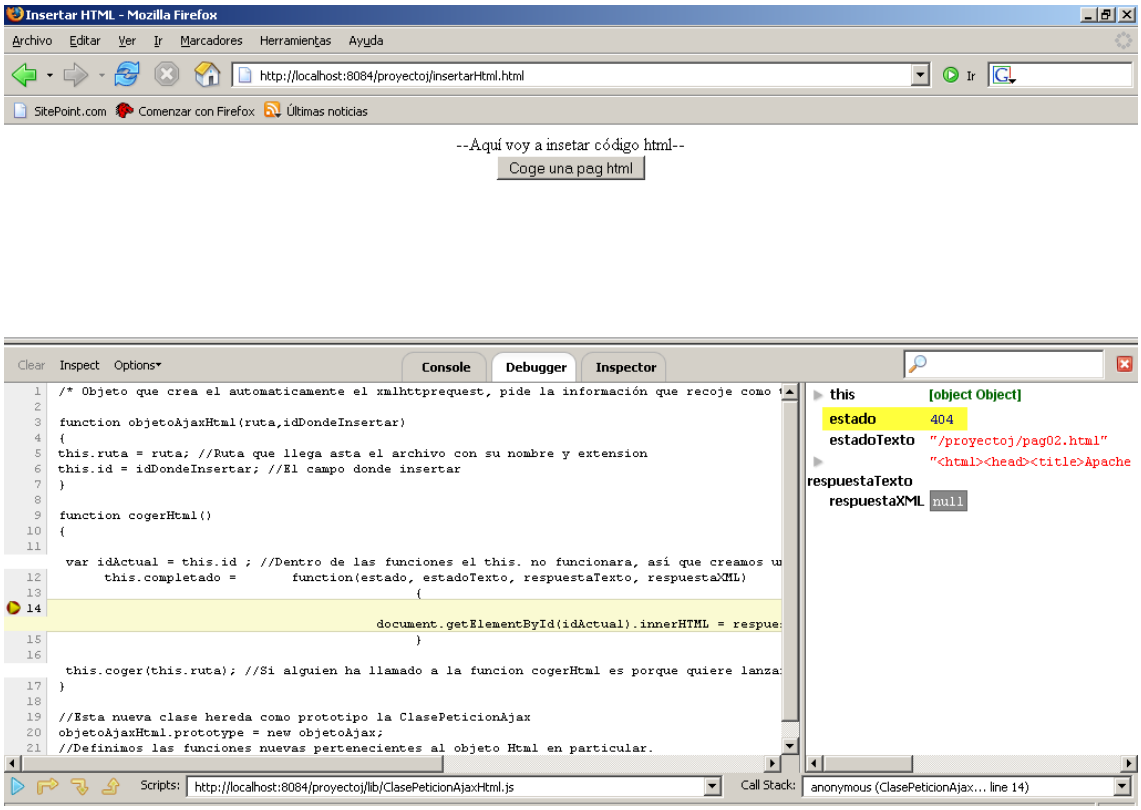


Ilustración 46 Aunque una página no se encuentre la clase AJAX actual la trataría igual.

Como vemos en la pequeña ventana de la derecha hemos recibido un código 404 y el servidor nos ha devuelto la típica página de error, ¡vamos a insertar la página de error!, parece que no es muy grave, en verdad nos vamos a enterar del error perfectamente, pero quedaría más bonito que saliera una alarma dándonos la misma información y no ocurriera nada, además existen más códigos de error menos típicos. Hasta ahora todo lo que hemos hecho si funciona correctamente devuelve el código 200, por lo que cualquier cosa que no sea 200 ahora mismo la tomamos como extraño.

Una pequeña detección de errores sería la siguiente (como el código es largo se han puesto puntos suspensivos en las partes innecesarias):

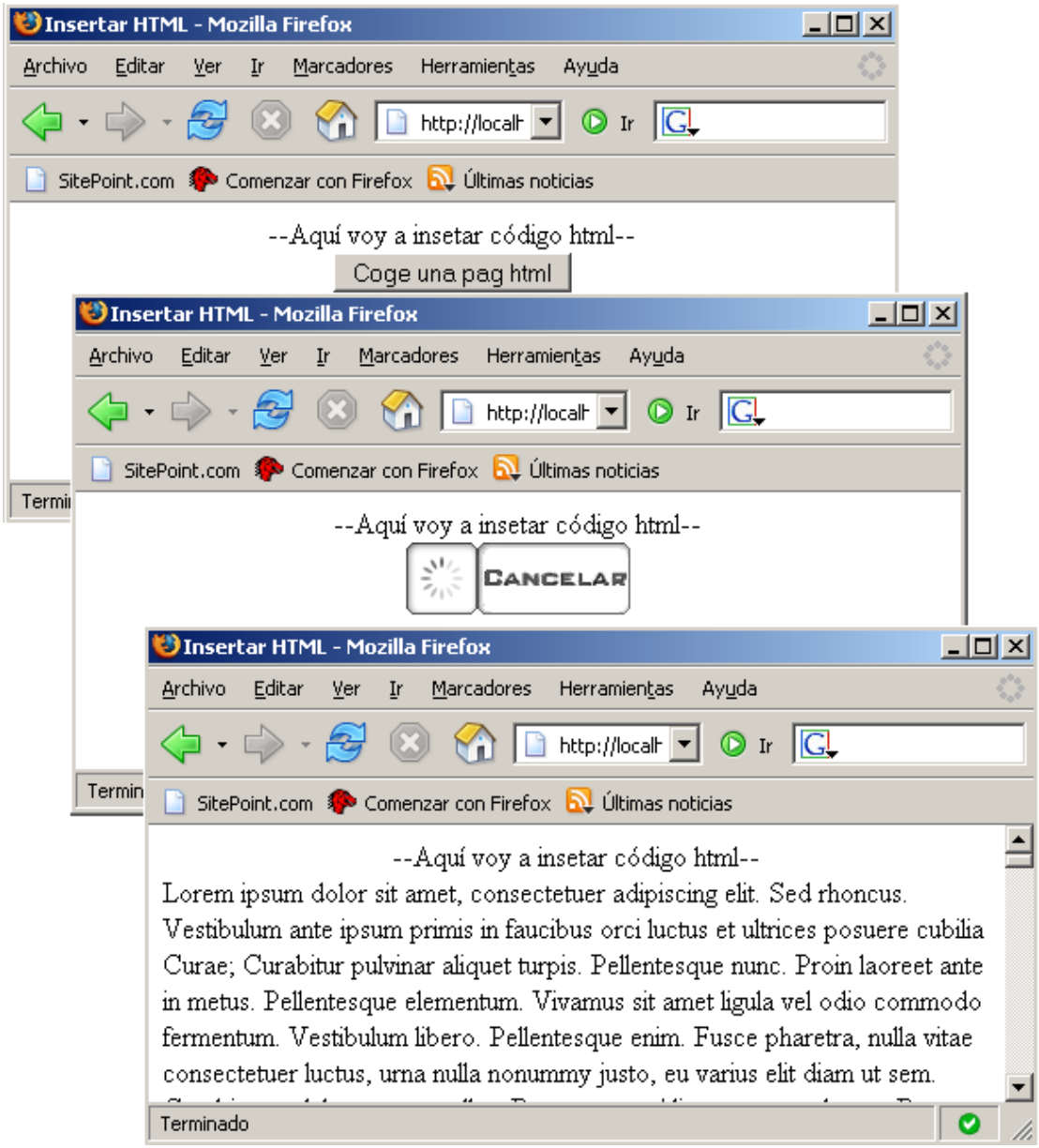
```
function peticionAsincrona(url) { //Función asignada al método coger
del objetoAjax.
...
      this.objetoRequest.onreadystatechange =
            function()
            {
                  switch(objetoActual.objetoRequest.readyState)
                  {
                        case 1: //Función que se llama cuando se está
cargando.
                        objetoActual.cargando();
                        break;
                        case 2: //Función que se llama cuando se a
cargado.
                        objetoActual.cargado();
                        break;
                        case 3: //Función que se llama cuando se está
en interactivo.
                        objetoActual.interactivo();
                        break;
                        case 4:
/*Detección de errores, solo nos fijamos en el codigo que nos llega

normalmente como bueno, como por ahora no es necesario elevar la

complejidad de la detección la dejamos así. */
                              if(objetoActual.objetoRequest.status !=
200)
                              {
                                    alert("Posible Error: " +
objetoActual.objetoRequest.status + ", Descripción: "

+ objetoActual.objetoRequest.statusText);
                              //Por si queremos hacer algo con el error
                              manejadorError(objetoActual.objetoRequest.s
                              tatus);
                                    }
                                    else //Si no hubo error, se deja al
programa seguir su flujo normal.
                                    {
/*Función que se llama cuando se completo la transmisión, se le envían
4 parámetros.*/
      objetoActual.completado(objetoActual.objetoRequest.status,
                        objetoActual.objetoRequest.statusText,
                        objetoActual.objetoRequest.responseText,
                        objetoActual.objetoRequest.responseXML);
                     }
                        break;
                  }
            }
      this.objetoRequest.send(null); //Iniciamos la transmisión de
datos.
}
//Declaramos los manejadores
function objetoRequestCargando() {}
...
```

La detección de errores está enmarcada en amarillo para que se vea más claro, es un pequeño cambio pero uno que nos dará información sin necesidad de estar utilizando un depurador, sin contar que es bastante más elegante detectar un posible error que dejar ver lo que pasa y si encima queremos tratarlo de alguna manera, podemos hacerlo con un manejador que está por defecto en blanco y se puede redefinir.

3.9 Dar soporte al usuario.

Una de las cosas que debemos tener clara es que el usuario final, navegante, cliente, etc. que se encuentre con nuestra pagina Web o programa que use tecnología AJAX pulsará sus botones y esperara que ocurra "algo", y desesperará, por ello estaría bien que viera algo mientras espera, un barra de carga, un mono saltando, etc. depende de la naturaleza de nuestra aplicación.

Vamos a añadir a nuestra librería, exactamente al objeto de pedir código HTML una función que se llamará cuadroEstadoCarga, que no será obligatorio usarla y que se podrá redefinir en el código de la página, por si no nos gusta el que hacemos por defecto, pero que si quisiéramos usar, funcionará sin que tengamos que hacer nada más que decírselo, con los siguientes resultados.



He utilizado un texto un poco raro del que hay que dar crédito al creador de la librería isiAJAX en unos foros, usó este texto para uno de sus ejemplos de carga largos y podemos encontrar su trabajo en sourceforge, es un archivo de texto de

unos 20 Megas, si estas haciendo las pruebas en un servidor instalado en tu PC de casa es mejor que sea un archivo grande, si además es un archivo de texto y el navegador se ve obligado a procesarlo puede ayudar a que veas el cuadrito de carga, si no, pasará como un flash y no se apreciará.

La página principal del ejemplo anterior es la misma que en los ejemplos anteriores con los siguientes cambios:

insertarHtml.html

```
<html>
<head>
<title>Insertar HTML</title>
<script language="JavaScript" src="lib/ConstructorXMLHttpRequest.js">
</script>
<script language="JavaScript" src="lib/ClasePeticionAjax.js">
</script>
<script language="JavaScript" src="lib/ClasePeticionAjaxHtml.js">
</script>

<script language="JavaScript" type="text/javascript">
  var PeticionAjaxHtml01 = new
  objetoAjaxHtml("largo.txt","ParteHTML","PeticionAjaxHtml01");
</script>

</head>
<body>
<center>--Aquí voy a insetar código html--</center>
<span id="ParteHTML">
  <center>
    <button onclick="PeticionAjaxHtml01.cogerHtml()">Coge una pag html
    </button>
  </center>
</span>
</body>
</html>
```

Como ves hemos añadido un tercer componente a la llamada de la función, es optativo y debe ser nombre de la variable para que, cuando generemos el código HTML del cuadro cancelar que has visto en la imagen, podamos poner que la imagen del cuadro cancelar es un link al método abort del objeto XMLHttpRequest de la variable.

Para mostrar el cuadro lo único que hacemos es colocarlo en el lugar donde irá el texto ya que tenemos su "id" y cuando llegue el texto la función de completado volverá a sustituir automáticamente el código que hay en InnerHtml.

Explicado lo anterior, el código de la clase AJAX tendría la siguiente modificación:

ClasePeticionAjaxHtml.js

```
/* NombreVariable es el nombre de la variable que controla la
cancelación, no necesita ser la variable del objeto, de forma que un
objetoAjax podría cancelar a otro diferente. */
function objetoAjaxHtml(ruta,idDondeInsertar,nombreVariable) {
this.ruta = ruta; //Ruta que llega asta el archivo con su nombre y
extensión.
this.id = idDondeInsertar; //El campo donde insertar.
this.nombreVariable = nombreVariable; //Nombre de esta variable para
poner el cuadro de carga.
}
```

```
function cogerHtml() {
	var idActual = this.id ; //Dentro de las funciones el this. no
funcionará.
	this.completado = function(estado, estadoTexto, respuestaTexto,
respuestaXML)
			{
			document.getElementById(idActual).innerHTML =
respuestaTexto;
			}
	if (this.nombreVariable) //Sin nombre no hay cuadro, lo
mostramos antes de empezar la petición.
	{
	this.cuadroEstadoCarga(); /*Sacamos un icono de carga que
hipnotiza al usuario, de estas manera tardará mas en desesperar. */
	}

	this.coger(this.ruta); /* Si alguien ha llamado a la función
cogerHtml lanzamos la petición xmlhttprequest. */
}

function cuadroEstadoCarga()
{
	//Mostramos un cuadrito de carga en el lugar donde irá lo que
estamos cargando.
	document.getElementById(this.id).innerHTML =
			"<center>" +
	"<img src=\"lib/img/cargando.gif\" alt=\"load\" width=\"40\"
height=\"40\" />" +
	"<a href=\"javascript:" + this.nombreVariable +
".objetoRequest.abort();\">" +
	"<img border=\"0\" src=\"lib/img/cancelar.jpg\" alt=\"cancel\"
width=\"86\" height=\"40\">" +
	"</a>" +
	"</center>"
}

//Esta nueva clase hereda como prototipo la ClasePeticionAjax
objetoAjaxHtml.prototype = new objetoAjax;
//Definimos las funciones nuevas pertenecientes al objeto Html en
particular.
objetoAjaxHtml.prototype.cogerHtml = cogerHtml; //Le añadimos la
función cogerHtml.
objetoAjaxHtml.prototype.cuadroEstadoCarga = cuadroEstadoCarga; /* El
cuadro que indica el estado de la carga. */
```

Se ha señalado el código nuevo referente al cuadro de carga, el cuadro aparece por defecto centrado para dar un mejor aspecto como se ve en el código HTML generado, no es más que un par de imágenes y un link con muchas barras invertidas para quitarle a las comillas su sentido de carácter especial.

Igual que hemos hecho esto aquí se podría añadir a la petición de las imágenes de Tipo 1 pero, como no añade nada nuevo, no lo explicaremos ya que haríamos exactamente lo mismo; lo añadiríamos antes de que apareciera la imagen y luego ésta lo sustituiría.

Capítulo 4: Ejemplos reales de uso para AJAX

4.1 Descripción del capítulo.

Tras lo visto en el capítulo 3 se puede decir que se sabe manejar el objeto bastante bien, solo se ha dejado en el tintero un par de funciones del objeto, muy sencillas, éstas serán lo primero que se verá en el capítulo que nos ocupa.

Se solucionará el problema de los caracteres especiales en Internet, como los acentos, para que a la hora de transferir información se pueda hacer respetando la lengua de origen.

Tras esto se verán una serie de ejemplos típicos del uso que se le da normalmente AJAX en la web para que el lector se familiarice con su uso y se cree la base para emplear AJAX en futuros desarrollos.

Los ejemplos de este capítulo ya comienzan a ser considerablemente más grandes que en capítulos anteriores, sería bastante bueno que el lector probara los ejemplos y mirara su código completo en un bloc de notas que coloreé el código (como Notepad++) para que se situé mejor ya que se omitirán asiduamente las cabeceras y algún pedazo de código menos relevante aparecerá como puntos suspensivos alguna vez.

Otra cosa interesante es que en este capítulo se ha validado el código HTML siguiendo el estándar XHTML estricto marcado por consorcio world wide web, esto se ha hecho con una extensión para Mozilla Firefox llamada tidy.

4.2 La web actual.

Antes de empezar con las últimas pinceladas al uso del objeto XMLHttpRequest y ver los ejemplos de uso útiles se verá un primer ejemplo que se podría hacer haciendo uso de lo visto hasta ahora y que es más que interesante.

Ilustración 47 Esquema de una página web actual.

Las páginas web actuales tienen un modelo que siguen más o menos todas, por no decir que todas, si nos fijamos en la ilustración anterior veremos una página web de ejemplo en la que se han delimitado ciertas zonas.

Esta página tiene todas las zonas típicas que podemos encontrar en un portal web; aunque puede variar su posición y su aspecto gráfico su funcionamiento es igual, al menos siempre nos vamos a encontrar con un título(zona 4) y un menú(zona 5 de la), estos rara vez van a desaparecer de la página.

En cambio el cuerpo de la página(zona 1), va a cambiar dependiendo de la sección. Lo ideal sería que cuando cambiamos de sección solo variara esto y con lo que hemos aprendido hasta ahora es posible hacerlo.

Antes de AJAX se podían usar Frames, ahora la solución con AJAX es más elegante y permite hacer a los desarrolladores más virguerías.

4.3 Métodos GET, POST y caracteres especiales en Internet.

Se verán primero los métodos de forma teórica, seguido se solventará el problema de los caracteres especiales y se juntará todo en un ejemplo para que quede mucho más claro además de tener una base para copiar y pegar en un uso posterior.

4.3.1 Introducción a los métodos GET y POST.

Algunos lectores conocerán ya estos métodos con sus diferencias, bondades y malicias pero para los que no y ya que tenemos que adaptar nuestro objeto AJAX a su uso, veámoslos brevemente.

Método GET
Es el que hemos estado utilizando hasta ahora, sin discutir en ningún momento su uso, con este método la URL de la dirección pedida junto con los valores que se envían forman una misma cadena.

Ejemplo GET: Las 2 líneas forman 1 sola cadena de caracteres

```
https://localhost:8443/pruevas/servidorRespuestas.jsp?Nombre=Juan%20Ma
riano&Apellidos=Fuentes%20Serna&Cumple=29%20de%20Diciembre&TimeStamp=1
160758169187
```

Esto método tiene un inconveniente bastante malo, si te fijas en el historial de tu navegador tendrás muchísimas páginas que debido a que usan este método guardan junto con la dirección, las variables y su valor, si esto eran datos confidenciales estás perdido, pero si además pensamos que hay servidores que guardan un historial de las paginas web pedidas este tipo de información es ya un verdadero atentado contra la privacidad.

Por esto, lo primero que debemos cambiar es el uso del método GET al POST que se verá seguidamente, mencionar que por defecto la mayoría de toolkits AJAX usan el método GET, si usas alguno para desarrollar una aplicación fuérzalo a utilizar el método POST si es posible.

Método POST
Cadena de petición POST
`https://localhost:8443/pruevas/servidorRespuestas.jsp`
Cadena de valores de la petición POST

```
Nombre=Juan Mariano&Apellidos=Fuentes Serna&Cumple=29 de
Diciembre&TimeStamp=1160758191375
```

Todo queda mucho más separado y limpio, además se evita que quede en los historiales que es su punto fuerte, es lo que se recomienda utilizar, junto con un sistema de encriptación para asegurar el canal de comunicación.

4.3.2 Caracteres especiales.

Este problema surge debido a las diferentes codificaciones que hay para los caracteres en texto plano, si estas viendo una página que has hecho tu mismo de forma local no hay problema pero si es un servidor quien está generando información y utiliza otra codificación es casi seguro que los acentos se van a estropear por el camino y cuando se muestren al final serán unos signos de interrogación algo pintorescos y extraños.

Para solventar este problema haremos dos cosas, la primera, guardaremos las páginas web que se escriban en formato de texto UTF-8 que contiene todos los símbolos, una vez que hemos terminado la página web utilizando nuestro editor favorito si este no tiene opciones para guardar en formato UTF-8 no tenemos más que abrir el archivo HTML como si de un texto normal se tratase con el bloc de notas “en Windows”, irnos al menú de archivo->Guardar como…

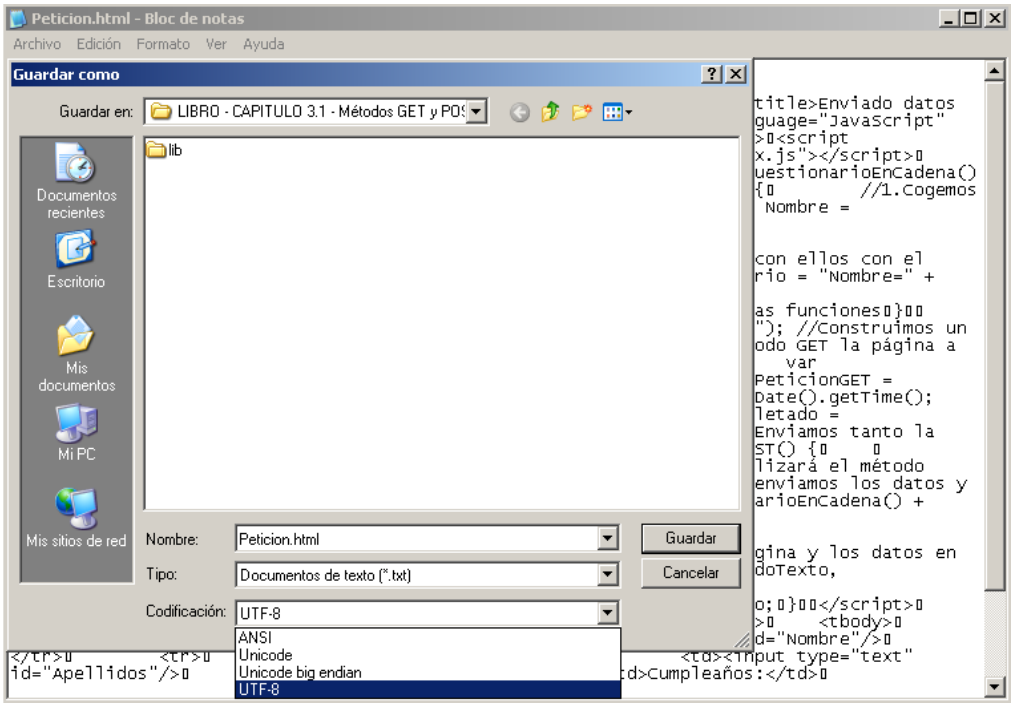


Ilustración 48 Como guardar en formato UTF-8 con el bloc de notas de Windows.

La segunda parte importante es que en la parte del servidor tenemos que decirle que el archivo está guardado también en ese formato (además de guardarlo) y cuando recojamos los parámetros enviados por el cliente hacerlo también aclarando que están en formato UTF-8, para esto añadiremos al archivo jsp las siguientes líneas que se pondrán correctamente en el ejemplo.

```
<%@ page contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
request.setCharacterEncoding("UTF-8");
```

4.3.3 Cambios en la librería para que acepte los 2 métodos.

Anteriormente utilizaba siempre el método GET, ahora con especificar cuál usar al crear el objeto y seleccionar más tarde el método con una estructura de selección simple será suficiente, remarcaremos lo interesante en el código y omitiremos lo que no sea necesario.

```
function objetoAjax(metodo)
{
…
this.metodo = metodo;
}
function peticionAsincrona(url,valores) //Función asignada al método
coger del objetoAjax.

// El parámetro valores sólo se usa en el caso POST
{
/*Copiamos el objeto actual, si usamos this dentro de la función que
asignemos a  onreadystatechange, no funcionara.*/
      var objetoActual = this;
      this.objetoRequest.open(this.metodo,  url,  true);  //Preparamos
la conexión.
…
  if (this.metodo == "GET")
  {
     this.objetoRequest.send(null);  //Iniciamos  la  transmisión  de
     datos.
  }
```

```
    else if(this.metodo == "POST")
  {
    this.objetoRequest.setRequestHeader('Content-
    Type','application/x-www-form-urlencoded');
    this.objetoRequest.send(valores);
  }
}
```

Como se aprecia, ahora cuando lancemos la petición si se utiliza el método POST debemos dar los valores en una cadena aparte, esto será suficiente para su buen comportamiento.

NOTA IMPORTANTE: El cambio hecho en este momento hace que si se quiere hacer uso de los objetos encapsulados del capítulo anterior junto con esta nueva versión del objetoAjax, se tengan que realizar ciertas modificaciones sencillas sobre ellos, este ejercicio se deja propuesto al lector interesado.

4.3.4 Ejemplo de uso de los métodos GET y POST.

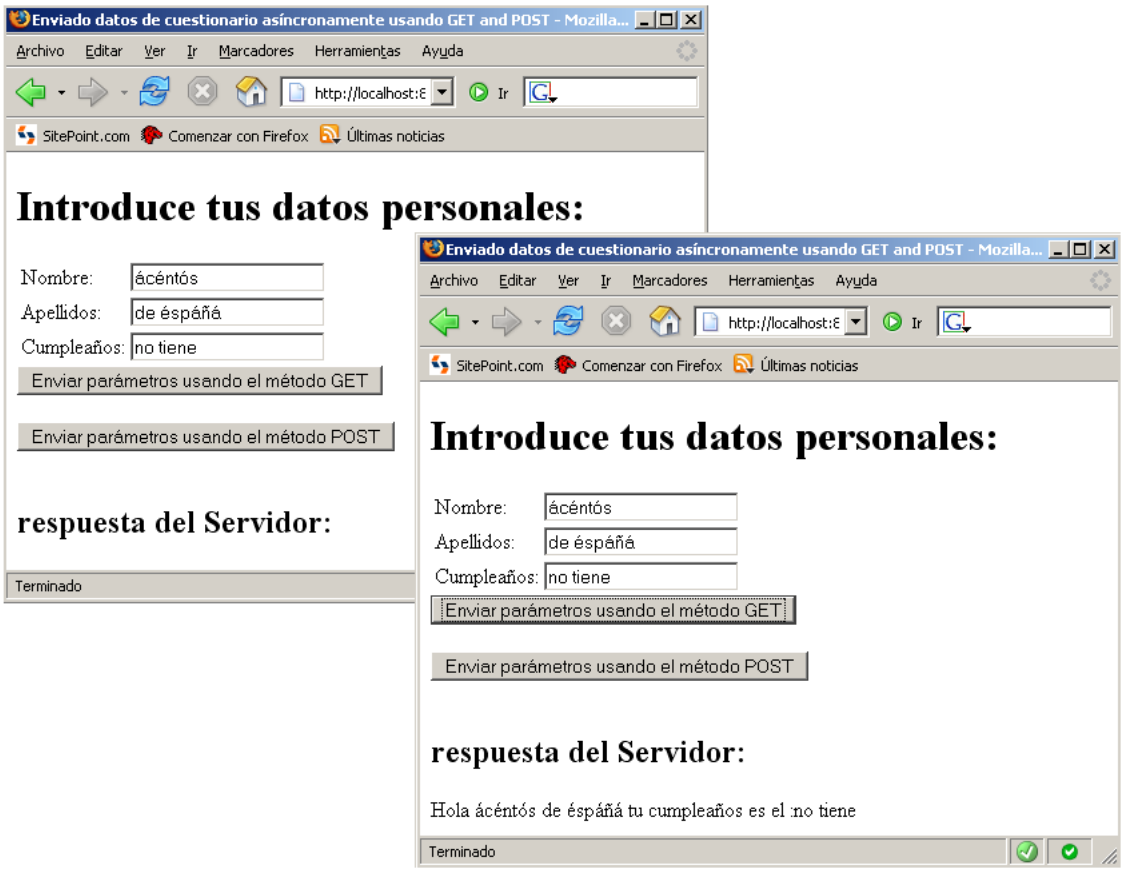


Ilustración 49 Ejemplo de los métodos GET y POST.

A la hora de escribir código en el lado del servidor no cambia nada el utilizar los métodos GET o POST como se ve en el código siguiente que no tiene nada especial para tratar la petición dependiendo del método usado:

servidorRespuestas.jsp

```
<!-- Para resolver los problemas con los acentos y la ñ debemos
añadir la siguiente directiva y guardar este archivo con codificación
UTF-8 con el bloc de notas, guardar como -->
<%@ page contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"
```

```
%>

<%
    request.setCharacterEncoding("UTF-8"); //Para resolver problemas
    con los acentos
    //Gracias al TimeStamp podemos hacer una comprovación de viveza si
    queremos
    //ej: Si los datos tienen mas de 5 segundos, no hacemos nada ya
    que puede ser un paquete viejo.
    String TimeStamp;
    TimeStamp = request.getParameter( "TimeStamp" );

    String Nombre;
    Nombre=request.getParameter( "Nombre" );

    String Apellidos;
    Apellidos=request.getParameter( "Apellidos" );

    String Cumple;
    Cumple=request.getParameter( "Cumple" );

    //Devolvemos una respuesta al usuario
    out.print("Hola " + Nombre + " " + Apellidos + " tu cumpleaños es
    el :" + Cumple);
%>
```

Sobre la parte HTML del ejemplo, lo más interesante es la diferencia a la hora de usar un método u otro ya que el método POST envía los valores en una cadena aparte de la url, el código fuente completo lo puedes encontrar en los códigos fuente del libro.

Peticion.html(solo parte del archivo)

```
…
function datosCuestionarioEnCadena() //Esta función construye una cadena con
los 3 datos
{
  //Cogemos los datos de cada campo y los metemos en una variable cada uno
  var Nombre = document.getElementById( "Nombre" ).value;
  var Apellidos = document.getElementById( "Apellidos" ).value;
  var Cumple = document.getElementById( "Cumple" ).value;
  //Construimos una cadena con ellos con el formato estándar de enviar
  información
  var cadenaPeticionCuestionario = "Nombre=" + Nombre + "&Apellidos=" +
  Apellidos + "&Cumple=" + Cumple;

  return cadenaPeticionCuestionario; //Devolvemos la cadena que se usara en
  otras funciones
}

function peticionUsandoGET()
{
  darInfo01= new objetoAjax("GET"); //Construimos un objetoAjax que utilizará
  el método GET
  /*Cuando se usa el método GET la página a la que enviamos los datos y los
  datos van unidos en la misma cadena */
  var cadenaPeticionGET = "servidorRespuestas.jsp?"; //La página.
  cadenaPeticionGET = cadenaPeticionGET + datosCuestionarioEnCadena() +
  "&TimeStamp=" + new Date().getTime(); //Unimos la página con el resto de los
  datos.

  darInfo01.completado = objetoRequestCompletado01;
  darInfo01.coger(cadenaPeticionGET); //Enviamos tanto la página como los
  datos en la misma cadena.
}

function peticionUsandoPOST()
{
```

```
  darInfo01= new objetoAjax( "POST" ); //Construimos un objetoAjax que
  utilizará el método POST
  //Cuando se utiliza el método POST la página a la que enviamos los datos y
  los datos van en cadenas diferentes.
  //Cadena de los datos
  var datosPOST = datosCuestionarioEnCadena() + "&TimeStamp=" + new
  Date().getTime();
  darInfo01.completado = objetoRequestCompletado01;
  //Enviamos la página y los datos en cadenas diferentes.
  darInfo01.coger( "servidorRespuestas.jsp" , datosPOST);
}
…
```

Con esto hemos terminado el vistazo a los ejemplos GET y POST además de solucionar el problema de los caracteres especiales.

4.4 Leer las cabeceras del objeto XMLHttpRequest.

Este punto nos servirá para dos cosas, para ver cómo leer las cabeceras del objeto XMLHttpRequest y para, según qué cabeceras nos lleguen, actuar de una forma u otra. Es bueno reutilizar los objetos y facilita muchos problemas a veces, si se pudiera hacer una petición y que según una señal que la respuesta se tratase de una forma u otra sería algo muy útil, esto se puede hacer mucho mejor y se hará, haciendo uso de XML, por ahora vamos a ver el ejemplo usando las cabeceras.

Es bastante simple, hacemos una petición como cualquier otra y leemos sus cabeceras, leemos una u otra dependiendo de una variable que se cambia a causa del botón que pulsamos.

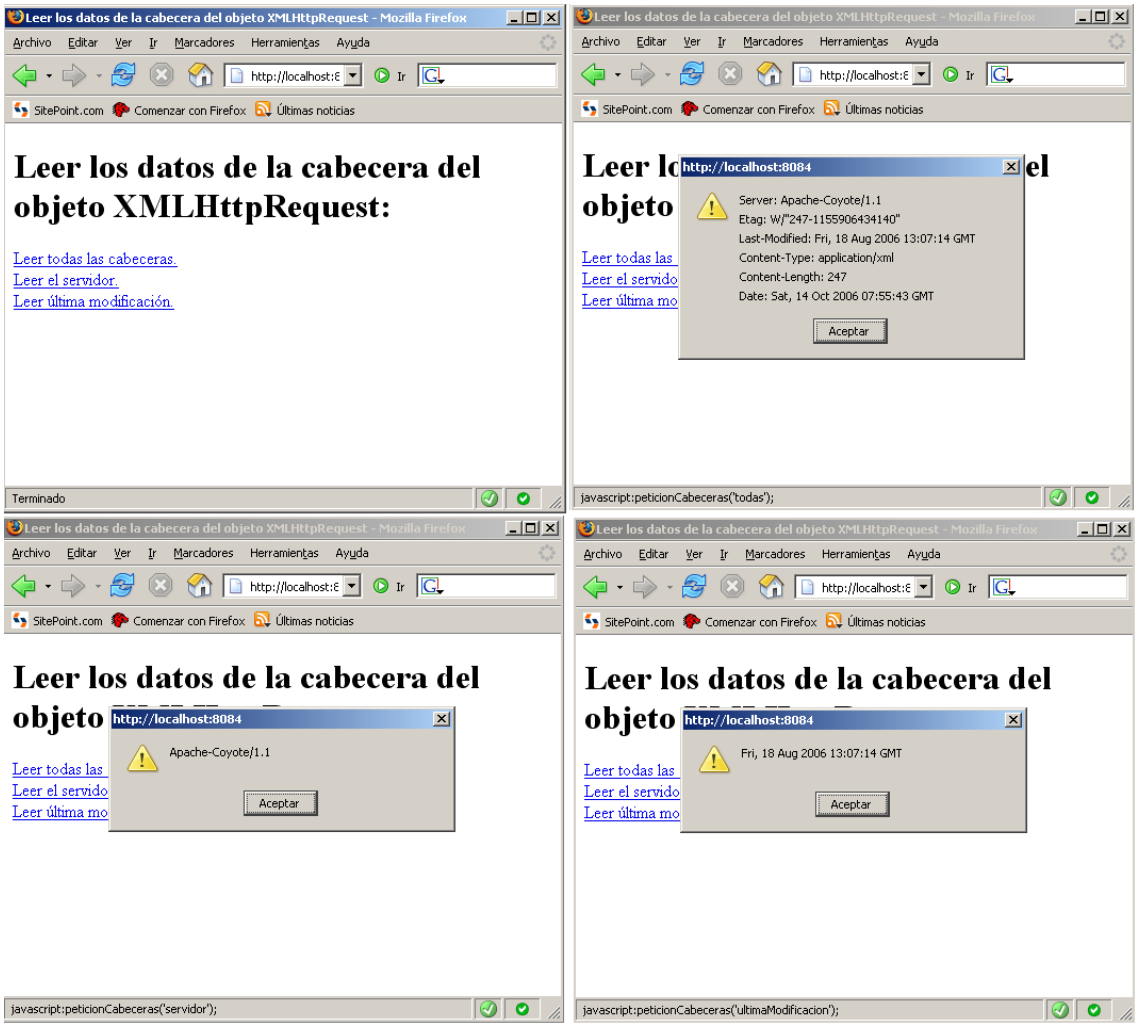


Ilustración 50 Ejemplo de leer las cabeceras del objeto XMLHttpRequest.

Como el archivo que recibimos en la petición no se usa en este caso prescindimos de el.

cabecerasDeLaRespuesta.html

```
<html>
<head>
<title>Leer los datos de la cabecera del objeto
XMLHttpRequest</title>
<script language="JavaScript" type="text/javascript"
src="lib/ConstructorXMLHttpRequest.js"></script>
<script language="JavaScript" type="text/javascript"
src="lib/ClasePeticionAjax.js"></script>
<script language="JavaScript" type="text/javascript">

function peticionCabeceras(cabecera)
{
var quecabecera = cabecera;
var recogerInfo01 = new objetoAjax("GET"); //Construimos un
objetoAjax que utilizará el método GET
recogerInfo01.completado =

function objetoRequestCompletado01(estado, estadoTexto,
respuestaTexto, respuestaXML)
{
  if(quecabecera == "todas")
  {
  alert(recogerInfo01.objetoRequest.getAllResponseHeaders());
```

```
  }
  else if(quecabecera == "servidor")
  {
  alert(recogerInfo01.objetoRequest.getResponseHeader("Server"));
  }
  else if(quecabecera == "ultimaModificacion")
  {
  alert(recogerInfo01.objetoRequest.getResponseHeader("Last-
  Modified"));
  }
}

recogerInfo01.coger("videoclub.xml"); //Enviamos tanto la página como
los datos en la misma cadena.
}
</script>
</head>

<body>
  <h1>Leer los datos de la cabecera del objeto
XMLHttpRequest:</h1>
  <a href="javascript:peticionCabeceras('todas');">Leer todas las
cabeceras.</a>
  <br />
  <a href="javascript:peticionCabeceras('servidor');">Leer el
servidor.</a>
  <br />
  <a href="javascript:peticionCabeceras('ultimaModificacion');">Leer
última modificación.</a>
</body>
</html>
```

Como podrá apreciar el lector si se fijó en la figura 4.4 para pedir una cabecera sola hay que utilizar el nombre por el cual está llamada cuando las pedimos todas.

4.5 Auto verificación y rendimiento en AJAX.

La auto verificación es una de esas cosas sencillas que puede ayudar muchísimo al usuario y hacer que nuestro portal tenga algo diferente respecto de otros, pero también puede ser una manera de cargar tanto el servidor que tengamos que comprar uno 10 veces mas grande.

Consiste básicamente en que ciertas acciones, como podría ser ver si una página Web existe o un usuario existe, se realicen sin necesidad de que el usuario tenga que pulsar ningún botón. Estas acciones se realizan automáticamente cuando fueran útiles, evitando muchas veces cargar una página de error, el ejemplo que nos ocupa trata una hipotética creación de usuario, estoy seguro de que más de un lector ha tenido problemas para registrarse debido a que los nombres de usuario que le gustaban estaban ya en uso, este ejemplo intenta resolver esto de forma más elegante sin tener que recibir una página nueva que diga “el nombre de usuario esta cogido”, con la consecuente pérdida de tiempo.

El ejemplo siguiente consta de 3 archivos, uno .html que tiene tanto código JavaScript como HTML y dos archivos .jsp, el único archivo que no es pequeño es el html que se dividirá en dos partes para mostrarlo en este texto, dicho esto comenzamos con el ejemplo.

autoVerificar.html (Parte HTML)

```
<body onload="desconectaBoton(1)">
```

```
<h1>EJEMPLO DE VERIFICACION DE USUARIOS USANDO AJAX</h1>
<br />
Tenemos una hipotética base de datos (simulada) solo con dos
usuarios, JuanMa y Sergio.
<br />
Queremos crear un nuevo usuario.
<br />
El sistema comprobará mientras que escribes si tu nombre de
usuario esta ya en la base de datos o no.
El nombre debe ser de al menos 4 caracteres de largo.
<br />

<form action="insertar.jsp" method="get">

  <table border="0">
  <tr>
  <!--Si usamos el atributo id, la variable idUsuario no se enviara a la
  pagina insertar.jps y si usamos name no referenciaremos el objeto
  dentro de esta página, solución, usar los 2 -->
    <td>
      <input type="text" id="idUsuario" name="idUsuario" size="20"
      onkeyup="validarUsuario()" />
    </td>
    <td>
      <span id="mensajeValidacion">
      </span>
    </td>
  </tr>
  </table>

  <input type="Submit" id="botonAceptacion" value="Crear Cuenta" />
</form>

</body>
```

Como se ve es un formulario muy sencillo, se destaca la primera línea debido a que tiene una cosa algo curiosa, nada más terminar de cargar la página se lanza una función, esta función desconectará el botón ya que éste se activará y desactivará dependiendo de si el usuario introducido es válido o no, el aspecto del ejemplo es el mostrado en la ilustración que sigue.

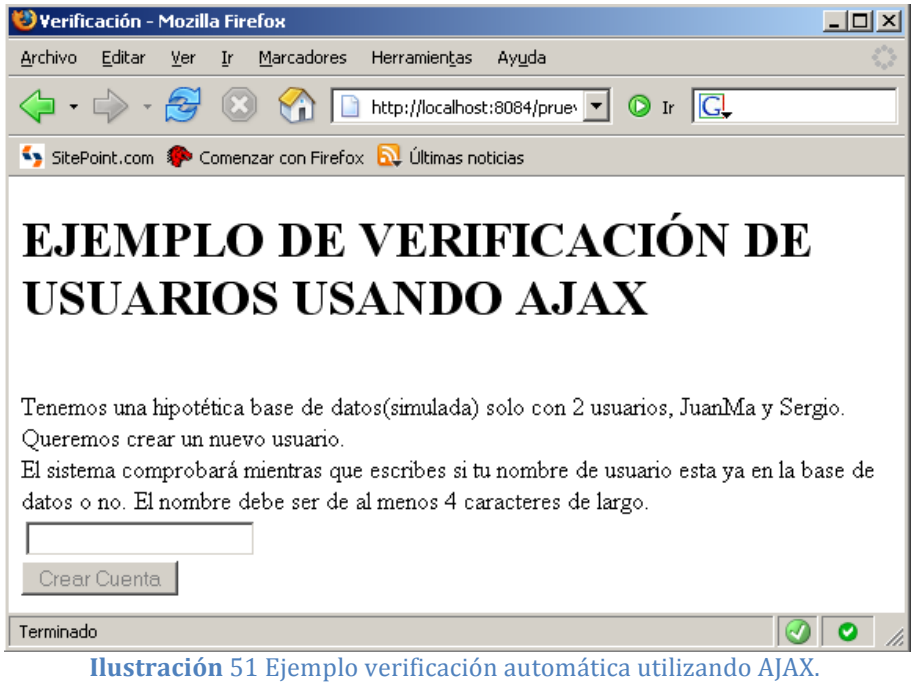


Ilustración 51 Ejemplo verificación automática utilizando AJAX.

autoVerificar.html (Parte Javascript)

```
<script language="JavaScript" type="text/javascript">
var PeticionAjax01 = new objetoAjax("GET"); //Definimos un nuevo
objetoAjax.
PeticionAjax01.completado = objetoRequestCompletado01; //Función
completado del objetoAjax redefinida.

function validarUsuario()
{
  if (!posibleUsuario) //Crea una variable con el nombre del Posible
  nuevo usuario
  {
  var posibleUsuario = document.getElementById("idUsuario");
  }

  // Envía el nombre a la url de verificación si este es de al menos 4
  caracteres de largo.
  if (escape(posibleUsuario.value.length) > 3)
  {
  var url = "validarUsuario.jsp?id=" + escape(posibleUsuario.value);
  PeticionAjax01.coger(url);
  }
  else
  {
  desconectaBoton(1); //Desactivamos el botón si el nombre es muy
  corto.
  document.getElementById('mensajeValidacion').innerHTML = "Nombre muy
  corto."; //Si había texto lo borramos.
  }
}

function objetoRequestCompletado01(estado, estadoTexto,
respuestaTexto, respuestaXML)
{
  //La pagina jsp devolverá true si el nombre se puede usar y false si
  ya está usando por otro usuario.
  if(respuestaTexto == "true")
  {
  document.getElementById('mensajeValidacion').innerHTML = "<div
  style=\"color:green\">Nombre usuario libre.</ div>";
  desconectaBoton(0); //Si se puede usar mantenemos el botón activado.
  }
  else
  {
  document.getElementById('mensajeValidacion').innerHTML = "<div
  style=\"color:red\">Nombre de usuario ya cogido.</ div>";
  desconectaBoton(1); //Si no se puede usar desactivamos el botón.
  }
}

function desconectaBoton(opcion)
{
    var boton = document.getElementById("botonAceptacion");
    boton.disabled = opcion;
}
</script>
```

La idea es muy sencilla, la función validarUsuario() envía el nombre al servidor al archivo validarUsuario.jsp si éste es de al menos de cuatro caracteres, el servidor sólo nos puede contestar true(el nombre se puede usar) o false(nombre ocupado). Dependiendo de esta respuesta que se recibe en la función objetoRequestCompletado01() se deja el botón activo o se desactiva, de esta manera al usuario no le llegará la típica página diciendo que su nick de registro

está ya cogido cada vez que intenta registrarse y se evita con ello perder el contexto y los datos de la página donde estamos.

Con lo anterior explicado los estados posibles en los que puede estar la página son los mostrados en la ilustración siguiente:

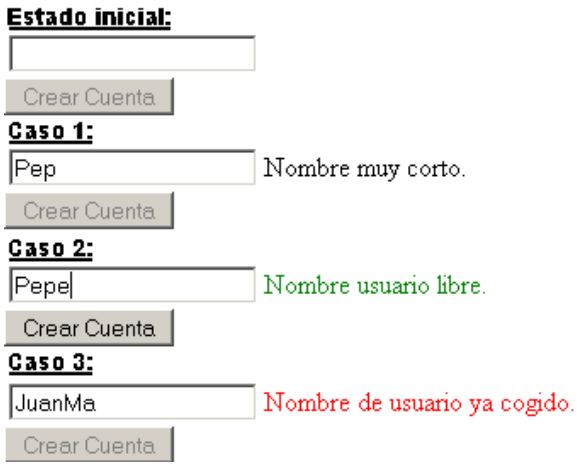


Ilustración 52 Ejemplo de verificación, estados posibles.

Aunque no añade mucho al ejemplo aquí están los archivos jsp ya que pueden aclarar de dónde se saca el valor de respuesta y cómo se ha simulado el mirar en una base de datos.

validarUsuario.jsp

```
<%
String usuario;
usuario=request.getParameter("id");
  //Tenemos una hipotética base de datos solo con JuanMa y Sergio, la
  simulamos de esta manera.
  if ( (usuario.equals("JuanMa")) || (usuario.equals("Sergio")))
  {
  out.print("false");
  }
  else
  {
  out.print("true");
  }
%>
```

insertar.jsp

```
<html>
<head><title></title></head>
<body>
<!--Simulamos la inserción del usuario en la base datos, pero no es real, no se
inserta nada. -->
Usuario insertado: <%= request.getParameter("idUsuario") %> <br />
<a href="index.html">Vuelve a la página principal</a>.
</body>
</html>
```

Esta, como muchas otras cosas que se pueden realizar haciendo chequeos consecutivos al servidor empleando AJAX, tienen un coste en términos de CPU y ancho de banda.

En algunos casos AJAX ayuda a ahorrar mientras que en otros hace que caiga el rendimiento considerablemente, este es uno de los segundos, cada vez que nuestro usuario libere una tecla se enviará una petición.

Si se llena una página de servicios como éste y el servidor ya estaba más o menos copado puede ser un gran problema, por ello el diseñador de la aplicación web debe tener estas cosas en cuenta y saber bien con qué está trabajando.

4.6 Pidiendo y analizando documentos XML.

En este apartado se va a iniciar al lector en el uso de archivos XML para recibir una respuesta, analizarla y mostrar el resultado en pantalla.

Recorrer el DOM de un archivo XML y sacar información es similar a hacerlo en un documento HTML, aunque en la práctica es mucho más sencillo, ya que no "suelen" tener la complejidad del DOM del navegador, y en esto es donde se centra el ejemplo.

En capítulos posteriores se verá cómo generar dinámicamente el archivo XML desde el servidor, por ahora se comenzará pidiendo un archivo ya existente que se tratará haciendo uso de JavaScript desde la página web principal, este ejemplo pues, solo tendrá dos archivos significativos, el archivo .xml y la página web .html que contendrá el código JavaScript necesario, como siempre, se seguirá haciendo uso de la librería AJAX construida anteriormente.

El lector podrá notar que no se hace uso de las clases encapsuladas (ClasePeticionAjaxHtml.js, ClasePeticionAjaxImagen.js, ClasePeticionAjaxJavascript.js) en este capítulo, si el lector revisa el apartado donde se explicaban los métodos GET y POST, verá que se hizo uso del ObjetoAjax aunque hubiera sido posible evitarlo realizando las modificaciones oportunas a ClasePeticionAjaxHtml.js.

Entonces no se hizo así ya que había que hacer cambios en la librería y es más instructivo verlo en el objeto mas genérico, pero en el caso del ejemplo que nos ocupa tenemos que definir un manejador diferente al que esta clase aporta y ya que no se ahorra trabajo, se ha preferido utilizar el objeto principal directamente.

Ya que un documento XML contiene información guardada de una manera estructurada y lo que se pretende es rescatar esta información del servidor, filtrar la parte útil y mostrarla dinámicamente a causa de una petición del usuario, haremos esto rescatando un archivo XML de un hipotético videoclub, haciéndolo lo mas sencillo posible.

El ejemplo es el siguiente:

videoclub.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<VideoClub>
      <Infantil>
```

```
            <Titulo>El rey León</Titulo>
            <Titulo>Aladín</Titulo>
      </Infantil>

      <Adultos>
            <Titulo>Eraser</Titulo>
            <Titulo>Instinto Básico</Titulo>
      </Adultos>
</VideoClub>
```

AnalizarXML.html (Parte HTML)

```
<body>
  <h1>Analizando una respuesta XML de un videclub.</h1>
  <br/><br/>
  <form action="#">
    <input    type="button"    value="Ver    todas    las    películas"
    onclick="comenzarPeticion('Todas');"/>
    <br/><br/>
    <input    type="button"    value="Ver    solo    las    infantiles"
    onclick="comenzarPeticion('Infantiles')"/>
  </form>
  <br />
  <span id="respuestaServidor"></span>
</body>
```

Antes de ver la parte JavaScript que es la compleja sería bueno que el lector se familiarizara con el resultado que se espera, que en este caso es muy sencillo y está mostrado en la figura siguiente.

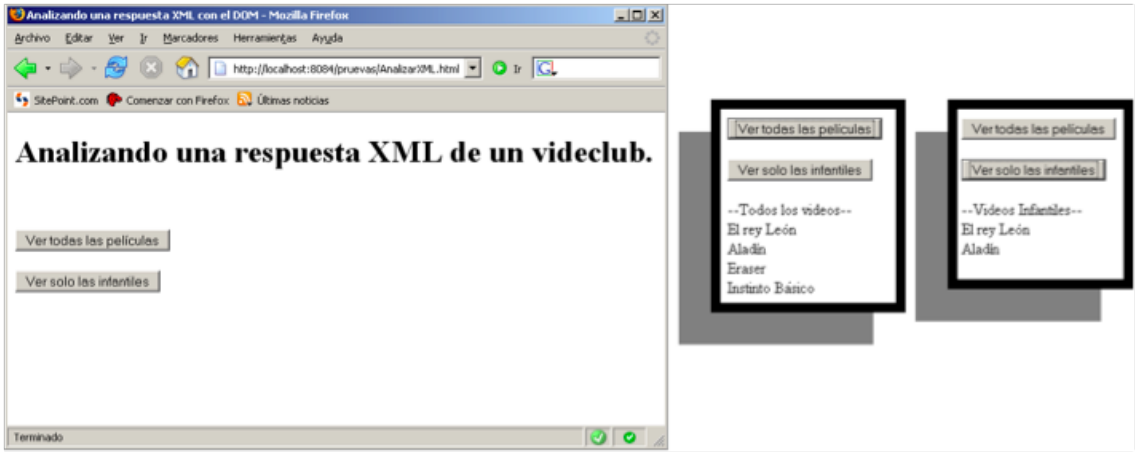


Ilustración 53 Resultado del análisis de un documento XML recibido con AJAX.

AnalizarXML.html (Parte JavaScript)

```
peticion01= new objetoAjax("GET"); //Construimos un objetoAjax que utilizará el
método GET

function comenzarPeticion(listaPedidaBoton) {
//Podemos tener varias opciones a la hora de mostrar la información.
var listaPedida = listaPedidaBoton ;
  peticion01.completado = function objetoRequestCompletado01(estado,
  estadoTexto, respuestaTexto, respuestaXML) {
    //Dependiendo de lo pedido debemos procesar la información de una manera u
    otra.
    if(listaPedida == "Todas")
    {
    listarTodo(respuestaXML);
    }
    else if (listaPedida == "Infantiles")
```

```
    {
    listarParte(respuestaXML);
    }
  }
peticion01.coger("videoclub.xml");
}

function listarTodo(documentoXML) {
//Cogemos todo lo que contienen las etiquetas título.
var documentoCompleto = documentoXML.getElementsByTagName("Titulo");
imprimirLista(" --Todos los videos-- ", documentoCompleto);
}

function listarParte(documentoXML) {
//Primero nos quedamos con la etiqueta Infantil.
var parte = documentoXML.getElementsByTagName("Infantil")[0];
var parteDeLaParte = parte.getElementsByTagName("Titulo");
//Luego con las etiquetas Título dentro de la etiqueta Infantil.
imprimirLista(" --Videos Infantiles-- ", parteDeLaParte);
}

function imprimirLista(titulo,informacion) {
salida = titulo; // Comenzamos poniendo el titulo, luego encadenaremos todo el
documento aquí.
var campoActual = null;
  for(var i = 0; i < informacion.length; i++) {
  /* Cogemos el nodo correspondiente a la iteración (puesta para clarificar,
  podríamos hacerlo de la variable información directamente). */
  campoActual = informacion[i];
  //Lo añadimos a la salida dejando un retorno de carro.
  salida = salida + "<br>" + campoActual.childNodes[0].nodeValue;
  }
/*Incrustamos la salida en la página, le hemos dado un formato plano para que
el ejemplo sea sencillo, pero podríamos crear una tabla con links recogiendo
mas capos e insertándolos, etc.. */
document.getElementById("respuestaServidor").innerHTML = salida;
}
```

El análisis del documento XML se hace de la forma mostrada anteriormente independientemente de lo complejo que el documento sea, si lo fuera más se tendría que ir pidiendo etiquetas dentro de otras etiquetas como se ha mostrado, gracias a esto se pueden desarrollar programas mas complejos y como el lector puede vislumbrar toda la dificultad de la parte AJAX de cualquier programa será crear los manejadores para enviar, recibir y manejar la información que se transmita, que con un poco de práctica se convertirá en una tarea repetitiva.

4.7 Refrescar la pantalla automáticamente.

Este ejemplo es para mostrar una función de JavaScript que puede tener una importancia crítica en el desarrollo de ciertos programas con necesidades de tiempo y muchas veces concurrentes, gracias a AJAX ahora a esto podemos sumar el coger información de lado servidor con lo cual las posibilidades son ilimitadas.

El ejemplo del uso del función setTimeout es bastante sencillo pero siempre es mejor ver el resultado antes para comprender mejor qué hace el código; básicamente se va a refrescar la pantalla automáticamente mostrando un mensaje que irá cambiando a otro dinámicamente, éstos están almacenados en el servidor.

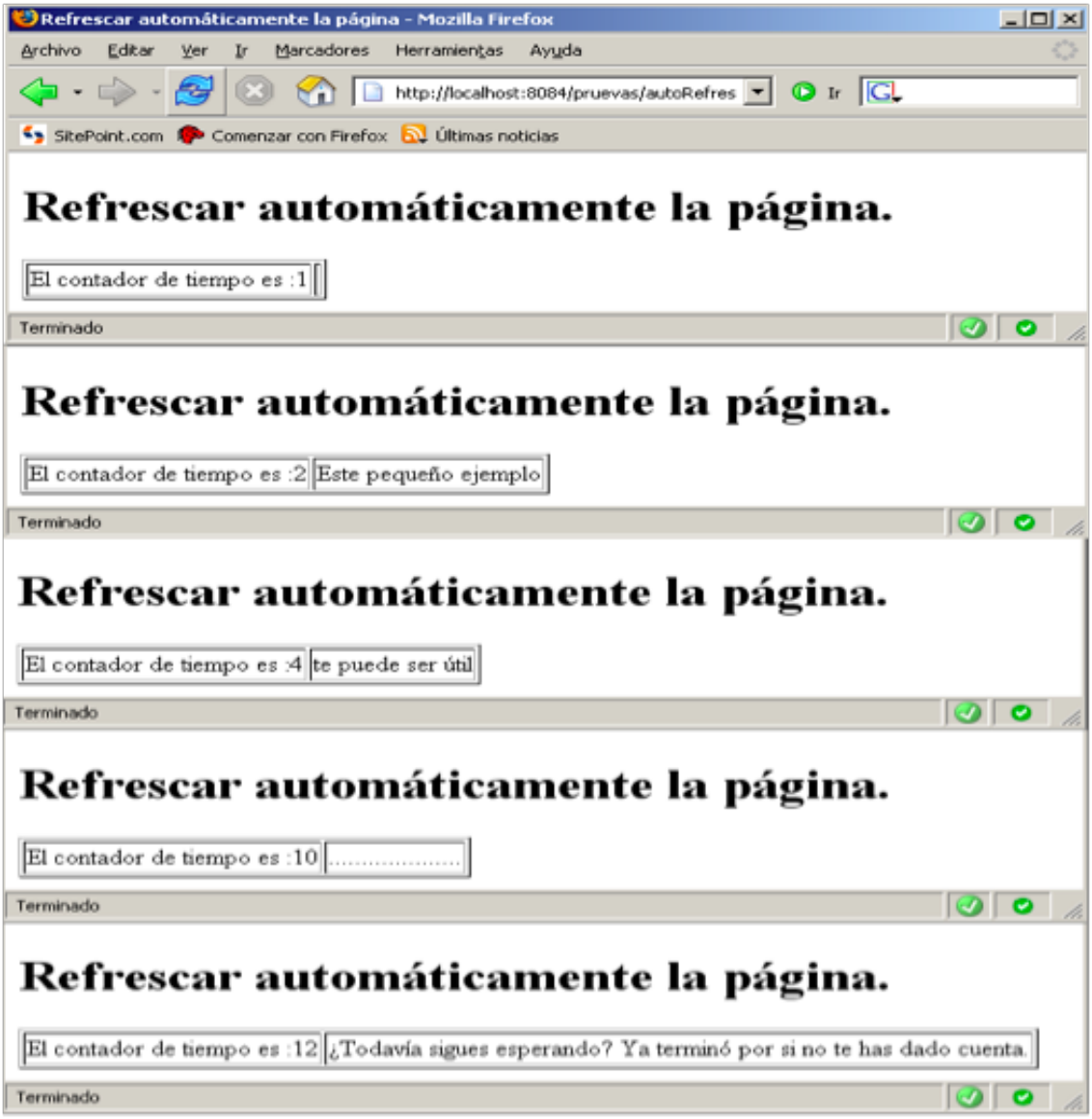


Ilustración 54 Ejemplo de refresco automático haciendo uso función setTimeout y AJAX.

El ejemplo tiene dos elementos que debemos destacar ya, el primero y más notable es que los cambios en la página se hacen sin permiso del usuario, él no dispara la acción. En principio hay quien puede pensar que esto es peligroso, y es verdad, lo es, pero también es poco evitable y muy útil para un desarrollador.

El segundo es que los mensajes son enviados por el servidor en este ejemplo, y se podrían generar de forma dinámica si hiciera falta. Claro está, en este ejemplo todo es muy simple.

El ejemplo consta solo de 2 archivos, bastante pequeños, seguimos haciendo uso de la misma librería AJAX creada anteriormente.

autoRefrescar.html

```
<html>
<head>
<title>Refrescar automáticamente la página</title>
<script language="JavaScript" type="text/javascript"
src="lib/ConstructorXMLHttpRequest.js"></script>
<script language="JavaScript" type="text/javascript"
```

```
src="lib/ClasePeticionAjax.js"></script>
<script language="JavaScript" type="text/javascript">
var tiempo = 0;

function refrescarAuto()
{
tiempo++;
document.getElementById("tiempo").innerHTML ="El contador de tiempo
es :" + tiempo;
  if((tiempo%2) == 0) //En los números pares (cada 2 segundos),
  hacemos la petición.
  {
  peticionFrase();
  }
setTimeout("refrescarAuto()", 1000); //Incrementamos la cuenta cada
segundo
}

function peticionFrase()
{
peticion01= new objetoAjax("GET"); //Construimos un objetoAjax que
utilizará el método GET
  peticion01.completado = function objetoRequestCompletado01(estado,
  estadoTexto, respuestaTexto, respuestaXML)
  {
  document.getElementById("respuesta").innerHTML = respuestaTexto;
  }
url = "cuentameAlgo.jsp?tiempo=" + tiempo;
peticion01.coger(url);
}
</script>
</head>
<body>
<h1>Refrescar automáticamente la página.</h1>
<table border="1">
      <tr>
            <td><span id="tiempo"></span></td>
            <td><span id="respuesta"></span></td>
      </tr>
</table>
<script language="JavaScript" type="text/javascript">
refrescarAuto();
</script>

</body>
</html>
```

cuentameAlgo.jsp

```
<!-- Para resolver los problemas con los acentos y la ñ debemos
añadir la siguiente directiva y guardar este archivo con codificación
UTF-8 con el bloc de notas, guardar como -->

<%@ page contentType="text/html; charset=UTF-8" pageEncoding="UTF-
8"%>

<%
//Para resolver problemas con los acentos a la hora de recoger la
información.
request.setCharacterEncoding("UTF-8");
int tiempo = 0;
tiempo = Integer.parseInt( request.getParameter("tiempo"));
      switch(tiempo)
      {
      case 2: out.print("Este pequeño ejemplo");
      break;
      case 4: out.print("te puede ser útil");
      break;
      case 6: out.print(".");
      break;
```

```
        case 8: out.print("...");
        break;
        case 10: out.print("....................");
        break;
        default: out.print("¿Todavía sigues esperando? Ya terminó por
si no te has dado cuenta.");
        }
%>
```

Como se puede apreciar el código es bastante sencillo, se hace una petición cada 2 segundos y se inserta el texto enviado por el servidor en el cuadro del cliente, las peticiones al servidor se hacen solo en los números pares con lo cual un número impar nunca llegará al default de la estructura caso del servidor.

Por último comentar que antes de AJAX se podía simular algo parecido teniendo una página .jsp que recibe la información de su construcción dinámicamente cada vez que se llama con lo que puede cambiar, como existe la posibilidad de hacer que la página completa se recargue cada cierto tiempo ésta se actualizaba, pero claro recargando la página entera haciendo que el servidor la genere de nuevo, ahora gracias a AJAX tenemos algo que con lo que utilizando menos recursos obtenemos más potencia.

Ejemplo recarga antes de poder coger información usando AJAX:

```
<META http-equiv= "refresh" content = "16;URL=index.jsp">
```

4.8 Una base de datos creada con el DOM y guardada con AJAX.

Antes de nada, para posicionarnos en el ejemplo, que nadie se engañe no vamos a hacer una interfaz SQL ni nada por el estilo, la idea es generar dinámicamente una tabla con información dada por el usuario con la siguiente interfaz, vista en la ilustración siguiente.

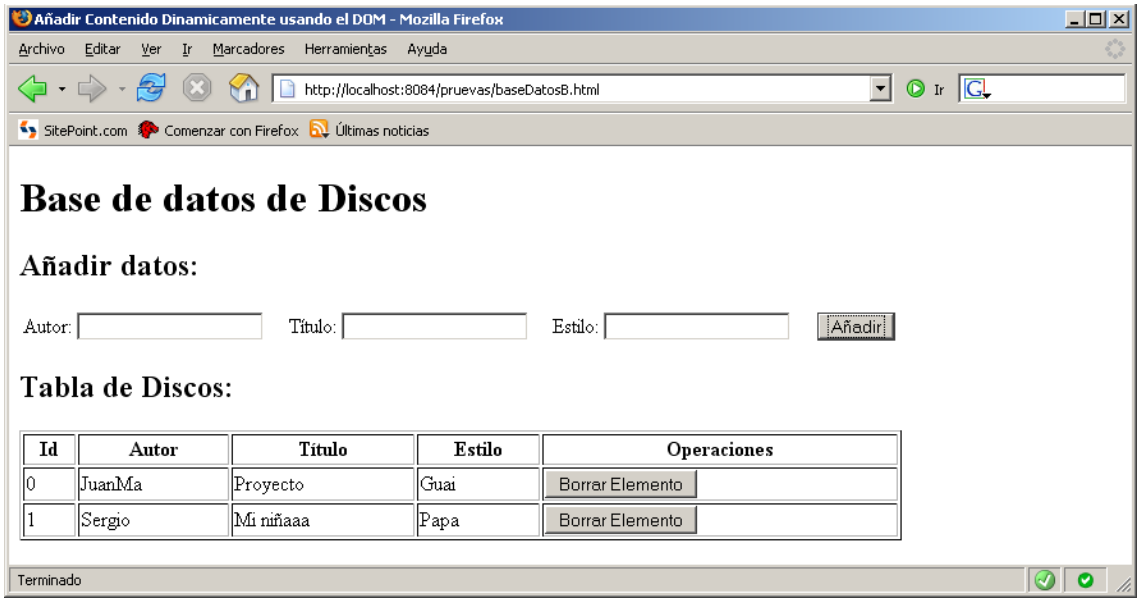


Ilustración 55 Tabla generada dinámicamente haciendo uso del DOM.

Esto no tiene nada que ver con AJAX, es solo una muestra de que se puede ir generando los elementos necesarios dinámicamente.

Aunque no lo parezca, es un gran avance ya que gracias a ello podemos realizar programas con un aspecto visual agradable rápidamente si los programamos en un web.

Pero necesitamos guardar la información que generamos, en otro caso no tendría ningún sentido. Llegado este momento podemos elegir archivo o base de datos, en este caso elegimos archivo, le enviaremos los datos a una página jsp que creará el archivo, de forma que la cosa gráficamente quedaría como se ve en la imagen siguiente.

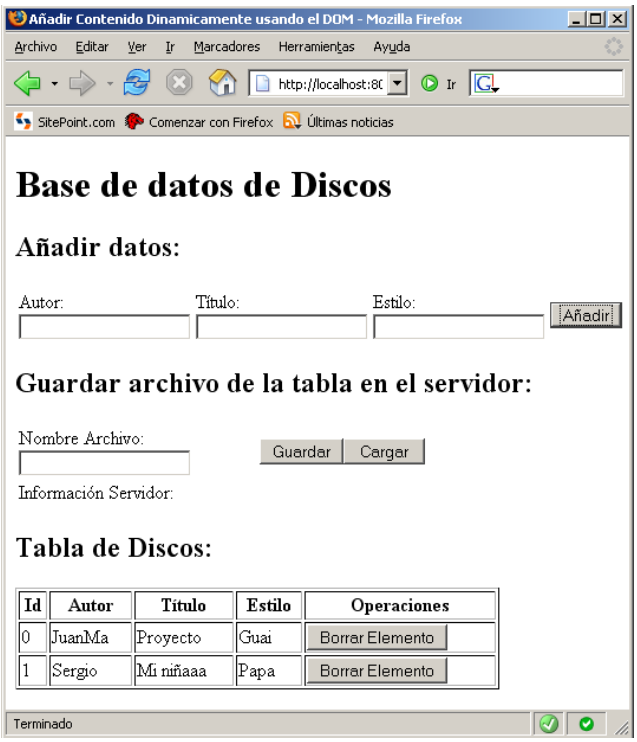


Ilustración 56 Interfaz simple para guardar y cargar de un archivo.

El ejemplo se ha intentado hacer lo más simple posible aun así su código es un poco largo, se ha generado la tabla haciendo uso del DOM por lo cual se complica un poco, pero en cambio guardamos en un archivo y rescatamos la información haciendo uso de la propiedad "innerHTML" lo que simplifica muchísimo esta parte. En otras palabras, el archivo almacena la información directamente en HTML pues sería una redundancia crear nuestro propio formato XML disponiendo ya de uno: el HTML.

4.8.1 Crear una tabla dinámicamente.

Comenzaremos mostrando cómo alcanzar el resultado mostrado en la figura 4.9 que es generar la tabla jugando con el DOM, esto lo hacemos así porque para una persona que no está acostumbrada a generar código de esta manera ya es suficiente problema y un escollo grande que se debe salvar para hacer aplicaciones grandes, nuestro ejemplo consta de solo un archivo que contiene tanto el código HTML como JavaScript y como no estamos haciendo uso de AJAX no nos hacen falta las librerías.

El archivo está comentado y es suficientemente auto explicativo pero algo largo por lo cual se ha decidido partir en tres partes: dos partes JavaScript cada una con una función y una parte html.

baseDatosB.html(Parte 1 JavaScript)

```
var identificadorUnico = 0;

function nuevoDisco() //Añadimos un disco a la tabla.
{
//1.Recogemos los valores de entrada.
var autor = document.getElementById("autor").value;
var titulo = document.getElementById("titulo").value;
var estilo = document.getElementById("estilo").value;
  //2.Nos cercionamos de que tengan sentido.
  if(autor == "" || titulo == "" || estilo == "")
  {
   return;
  }
  else
  {
  //3.Limpiamos las entradas.
  document.getElementById("autor").value = "";
  document.getElementById("titulo").value = "";
  document.getElementById("estilo").value = "";
  //4.Creamos una nueva fila para la tabla, usando el DOM y le
  añadimos las celdas
  var fila = document.createElement("tr");
  fila.setAttribute("id",identificadorUnico);     //Le     damos     un
  identificador único para poder reconocerla.

  var celda1 = document.createElement("td");
  celda1.appendChild(document.createTextNode(identificadorUnico));
  fila.appendChild(celda1);

  var celda2 = document.createElement("td");
  celda2.appendChild(document.createTextNode(autor));
  fila.appendChild(celda2);

  var celda3 = document.createElement("td");
  celda3.appendChild(document.createTextNode(titulo));
  fila.appendChild(celda3);

  var celda4 = document.createElement("td");
  celda4.appendChild(document.createTextNode(estilo));
  fila.appendChild(celda4);

  //5.Creamos un botón para asignárselo a la fila
  var botonborrar = document.createElement("input");
  botonborrar.setAttribute("type", "button");
  botonborrar.setAttribute("value", "Borrar Elemento");

  /*Hay al menos 3 maneras de hacer lo siguiente, yo lo voy a guardar
  como formato texto, así podemos guardar el código html después en
  formato texto y que la referencia al id se guarde, sino se perderá
  */

  botonborrar.setAttribute("onclick",          "borrarDisco("          +
  identificadorUnico + ")");
  //6.Metemos el botón dentro de una celda y lo añadimos a la fila
  var celda5 = document.createElement("td");
  celda5.appendChild(botonborrar);
  fila.appendChild(celda5);
  //7.Aumentamos el valor del contador de los identificadores únicos.
  identificadorUnico++;
  //8.Actualizamos la tabla
  document.getElementById("tablaDiscos").appendChild(fila);
```

```
  }
}
```

baseDatosB.html(Parte 2 Javascript)

```
function borrarDisco(BorrarIdentificadorUnico) //Borramos un disco de
la tabla
{
var borrarFila = document.getElementById(BorrarIdentificadorUnico);
var tablaDiscos = document.getElementById("tablaDiscos");
tablaDiscos.removeChild(borrarFila);
}
```

baseDatosB.html(Parte 3 HTML)

```
<body>
<h1>Base de datos de Discos</h1>
<h2>Añadir datos:</h2>
<form action="#">
<table width="80%" border="0">
      <tr>
            <td>Autor: <input type="text" id="autor"/></td>
            <td>Título: <input type="text" id="titulo"/></td>
            <td>Estilo: <input type="text" id="estilo"/></td>
            <td colspan="3" align="center">
            <input            type="button"            value="Añadir"
onclick="nuevoDisco()" />
            </td>
      </tr>
</table>
</form>
<h2>Tabla de Discos:</h2>
<table border="1" width="80%">
    <tbody id="tablaDiscos">
                  <tr>
                        <th>Id</th>
                        <th>Autor</th>
                        <th>Título</th>
                        <th>Estilo</th>
                        <th>Operaciones</th>
                  </tr>
      </tbody>
</table>
</body>
```

El funcionamiento (que no el código JavaScript) es bastante simple, tenemos una función que añade filas y otra que las borra, éstas se llaman debido a los eventos disparados por los botones de añadir y borrar, como en los programas de toda la vida pero en formato web, lo siguiente sería no perder la información al salirnos del navegador.

4.8.2 Guardar información de innerHTML usando AJAX.

Una forma rudimentaria pero eficaz de guardar y cargar información si ésta estaba en formato HTML es guardar directamente la cadena de texto que hay dentro de la propiedad “innerHTML” en un archivo y la forma más sencilla de recuperarla es leer la cadena e introducirla dentro de la propiedad “innerHTML” directamente, por supuesto se guardará en el servidor, cosa que puede ser muy útil en una empresa mediana donde queremos que los comerciales vayan dejando los pedidos de los clientes nada más confirmarlos por si se necesita cargar algo y no se deje para el día siguiente, etc.

Es decir, con AJAX podemos guardar código generado dinámicamente sin necesidad de una base de datos y luego recuperarlo.

Ahora el ejemplo constará de 3 archivos, el ya visto con alguna modificación y 2 archivos .jsp uno que guarda una cadena en un archivo y otro que la carga. La idea es bastante sencilla, tal vez lo mas difícil de todo sea generar el archivo de texto y cargarlo si no se tiene mucha experiencia con Java, se verán primero las modificaciones a la página principal, se han añadido cuatro funciones JavaScript que son las siguientes y no se han modificado las anteriores, además se han incluido las librerías AJAX.

baseDatos.html(Modificaciones con respecto a baseDatosB.html, parte JavaScript)

```
function guardar()
{
  darInfo01= new objetoAjax("GET"); //Construimos un objetoAjax que
  utilizará el método GET

  //Cuando se usa el método GET la página a la que enviamos los datos y
  los datos van unidos en la misma cadena.
  var cadenaPeticionGET = "guardarDatos.jsp?"; //La página.
  var datos ="&guardarEnFormatoTexto=" +
  document.getElementById("tablaDiscos").innerHTML +
              "&nombreArchivo=" +
  document.getElementById("nombreArchivo").value;
  cadenaPeticionGET =cadenaPeticionGET + datos; //Unimos la página con
  el resto de los datos.

  darInfo01.completado =  function objetoRequestCompletado01(estado,
  estadoTexto, respuestaTexto, respuestaXML)
                  {
                  document.getElementById("resultadoGuardar").innerHTML
                  = respuestaTexto;
                  setTimeout("borrarResultado()", 2000);
                  }
  darInfo01.coger(cadenaPeticionGET); //Enviamos tanto la página como
  los datos en la misma cadena.
}

function borrarResultado()
{
document.getElementById("resultadoGuardar").innerHTML = "";
}

function cargar()
{
  cogerInfo02= new objetoAjax("GET"); //Construimos un objetoAjax que
  utilizará el método GET

  //Cuando se usa el método GET la página a la que enviamos los datos y
  los datos van unidos en la misma cadena.
  var cadenaPeticionGET = "cargarDatos.jsp?"; //La página.
  var datos ="&nombreArchivo=" +
  document.getElementById("nombreArchivo").value;
  cadenaPeticionGET =cadenaPeticionGET + datos; //Unimos la página con
  el resto de los datos.

  cogerInfo02.completado = function objetoRequestCompletado02(estado,
  estadoTexto, respuestaTexto, respuestaXML)
                    {
                    document.getElementById("tablaDiscos").innerHTML =
                    respuestaTexto;
                    actualizarIdUnico();
                    }
```

```
  cogerInfo02.coger(cadenaPeticionGET); //Enviamos tanto la página como
  los datos en la misma cadena.
}

function actualizarIdUnico()
{
// El mayor identificador siempre se encuentra al final, así cuando
cargamos un archivo estará en la última fila de la tabla.
var tabla = document.getElementById("tablaDiscos");
var ultimaFila = tabla.childNodes.length - 1;
identificadorUnico = tabla.childNodes[ultimaFila].id; //Le damos  el
valor del id de la última fila.
identificadorUnico++; //Lo adelantamos al siguiente valor, así su
valor ya será valido como nuevo ID.
}
```

Como podrás apreciar, si lo lees atentamente, no se ha hecho un gran control de errores, pero si guardamos correctamente se indica al usuario y luego el mensaje desaparece, esta es una pequeña utilidad de la función "setTimeout", seguimos con la parte HTML a la que le hemos añadido el siguiente código perteneciente a la tabla de cargar y guardar.

baseDatos.html (parte HTML)

```
…
<h2>Guardar archivo de la tabla en el servidor:</h2>
<form action="#">
<table width="80%" border="0">
      <tr>
            <td  width="40%">Nombre  Archivo:  <input  type="text"
id="nombreArchivo"/> </td>
            <td  width="40%"><input  type="button"  value="  Guardar  "
onclick="guardar()" />
                                                      <input
type="button" value="  Cargar  " onclick="cargar()" />
             </td>
      </tr>
      <tr>
            <td width="40%">Información Servidor: </td>
            <td width="40%"><span id="resultadoGuardar"></span></td>
      </tr>
</table>
</form>
…
```

Queda por ver las páginas .jsp que guardan y cargan de un archivo.

guardarDatos.jsp

```
<!-- Para resolver los problemas con los acentos y la ñ debemos
añadir la siguiente directiva y guardar este archivo con codificación
UTF-8 con el bloc de notas, guardar como -->
<%@ page contentType="text/html; charset=UTF-8" pageEncoding="UTF-
8"%>
<%@ page import="java.io.*" %>

<%
request.setCharacterEncoding("UTF-8"); /*Para resolver problemas con
los acentos a la hora de recoger la información.*/

try
{
      String cadenaParaEscribir =
request.getParameter("guardarEnFormatoTexto");
      String nombreArchivo = request.getParameter("nombreArchivo");
      //0.Lo siguiente es para hacer pruebas, úsalo en vez de usar
las 2 líneas de arriba.
```

```
	//String cadenaParaEscribir = "Hola Mundo." ;
	//String nombreArchivo = "HolaMundo.txt";

	//1.Preparamos lo que queremos en formato de cadena dentro del
buffer, se va almacenando.
	StringBuffer bufferSalida = new StringBuffer();
	bufferSalida.append(cadenaParaEscribir);


	//2.Preparamos el archivo para escribir.
	String camino = application.getRealPath("/" + nombreArchivo);
	File archivo = new File(camino);
	FileOutputStream cadenaSalida = new FileOutputStream(archivo);
	PrintWriter punteroEscritura = new PrintWriter(cadenaSalida);
/*
// Esto se hace por convención de estratificación.
	PrintWriter punteroEscritura = new PrintWriter(
				new FileOutputStream(

application.getRealPath("/"+nombreArchivo)
				)
	);
	punteroEscritura.print(cadenaParaEscribir);
	punteroEscritura.close();
*/

	//3.Escribimos el archivo.
	punteroEscritura.print(bufferSalida);

	//4.Limpiamos y cerramos lo que hemos estado utilizando.
	punteroEscritura.flush();
	punteroEscritura.close();
	cadenaSalida.flush();
	cadenaSalida.close();

	out.print("<div style=\"color:green\">El archivo se guardo
correctamente.</ div>");
}
catch (Exception e)
{
	out.print("<div style=\"color:red\">No se pudo guardar el
archivo.</ div>");
}

%>
```

Para que el código anterior funcione correctamente la aplicación tiene que tener permiso de escritura en el directorio, puede parecer obvio pero es importante ya que la configuración de un servidor real puede ser algo restrictiva, lo único que quedaría por ver es la carga de datos.

cargarDatos.jsp

```
<%@ page contentType="text/html; charset=UTF-8" pageEncoding="UTF-
8"%>
<%@ page import="java.io.*" %>

<%
request.setCharacterEncoding("UTF-8"); /* Para resolver problemas con
los acentos a la hora de recoger la información. */
try
{
	//1.Variables para el proceso de lectura.
	String nombreArchivo = request.getParameter("nombreArchivo");
	String cadenaSalida = "";
	String cadenaAuxiliar;
	//2.Preparamos el archivo para Leer.
	String camino = application.getRealPath("/" + nombreArchivo);
```

```
        FileReader archivoLectura= new FileReader(camino);
        BufferedReader cabezaLectora = new
BufferedReader(archivoLectura);

        //3.Leemos el archivo (Técnica de primera lectura adelantada).
        cadenaAuxiliar = cabezaLectora.readLine();
        while(cadenaAuxiliar != null)
                {
              cadenaSalida =  cadenaSalida.concat(cadenaAuxiliar);
                cadenaAuxiliar = cabezaLectora.readLine();
                }
        //4.Limpiamos y cerramos lo que hemos estado utilizando.
        cabezaLectora.close();
        //5.Devolvemos el resultado.
        out.print(cadenaSalida);
}
catch (Exception e)
{
        out.print("<div style=\"color:red\">No se pudo cargar el
archivo, hubo un error.</ div>");
}
%>
```

Como el lector podrá apreciar en el momento que se escribe un pequeño programa que pudiera tener una utilidad real el nº de líneas de código que tiene se dispara, pero peor todavía, nos estamos concentrando en AJAX y lo único que hemos hecho es enviar y recibir una cadena de texto, podríamos haber hecho un ejemplo mas pequeño para esto, seguro, pero es también objeto de este texto ofrecer algunas posibilidades de uso. Normalmente todos los ejemplos explicados son muy pequeños y realmente no hace falta más ya que enseñan lo necesario para desarrollar aplicaciones reales; como se ve en ésta, la complejidad no se haya en la parte AJAX sino en generar dinámicamente la tabla haciendo uso del DOM y en desarrollar la parte .jsp.

4.9 Dar información dinámicamente utilizando los eventos y el DOM.

En este ejemplo no vamos a utilizar AJAX, es más bien DHTML ya que no hacemos uso del objeto XMLHttpRequest, pero nos va a ser muy útil cuando construyamos programas más grandes ya que necesitamos entornos mas interactivos para que sean atractivos. El lector se habrá dado cuenta cuando abre un menú que, al desplazar el ratón sobre él, muchas veces aparecen nuevas opciones automáticamente o que si espera un par de segundos con el puntero del ratón sobre un botón de una barra de herramientas aparece una descripción de su utilidad, todo esto se ve en la imagen siguiente.

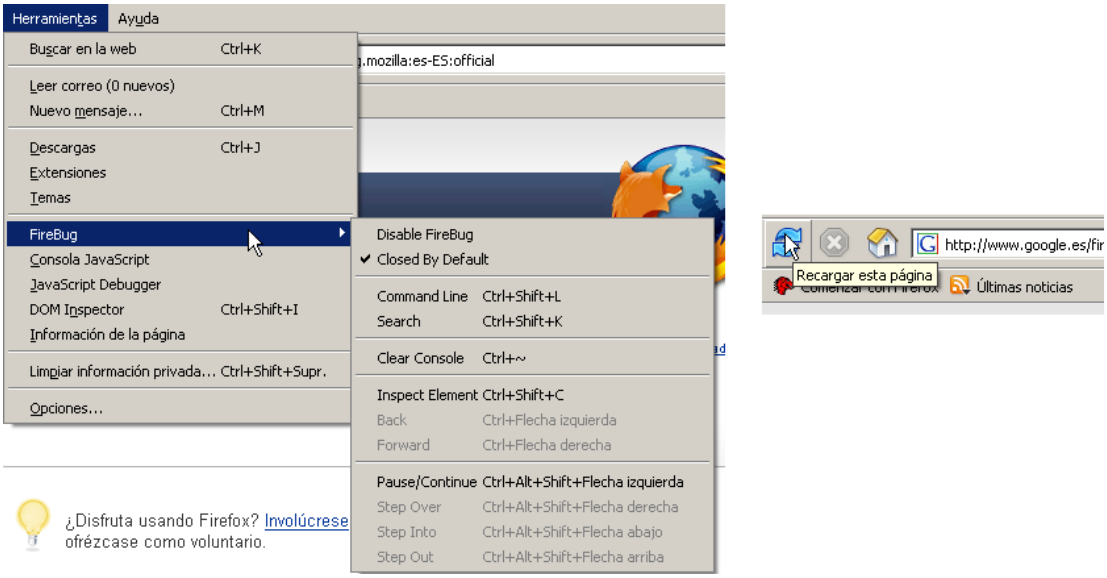


Ilustración 57 Menú típico de un programa actual en este caso las imágenes son de Mozilla Firefox.

Este tipo de cosas, por increíble que parezca, se pueden hacer en un entorno web de forma relativamente sencilla y es lo que veremos en el siguiente ejemplo ya que, una vez que sepamos generar tablas dinámicamente donde nos plazca, las posibilidades que se nos abren son muchas; entre otras se nos abre el campo de los videojuegos, no solo hacer barras de herramientas, ya que podemos mostrar cualquier gráfico como fondo de una tabla y mover la tabla y modificar el gráfico dinámicamente a causa de los eventos del teclado o ratón del usuario, esto unido a AJAX es potentísimo.

4.9.1 Ejemplo 1 – Tabla relativa a otra tabla.

Es muy importante cuando queremos posicionar un elemento relativo a otro comprender la forma en que se calculan las coordenadas, tenemos que pensar que trabajamos con etiquetas anidadas en HTML, esto lo menciono porque es muy importante cuando pedimos la posición de una tabla que se creó al cargar el documento la posición de la etiqueta de esta tabla que tiene sus propiedades y son relativas al padre. Es decir, para obtener su posición absoluta necesitamos como dirección base la del padre y sumarle el desplazamiento del hijo, como se ve en la figura siguiente.

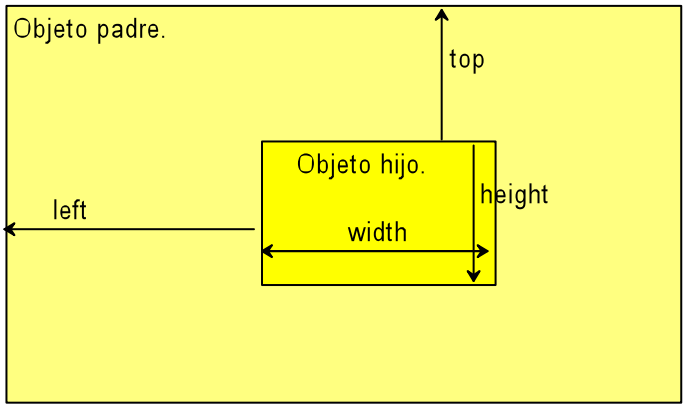


Ilustración 58 Las posiciones "suelen" ser relativas a otras.

Pero lo peor es que si el padre tiene otro padre el comportamiento es recursivo como muestra la ilustración siguiente.

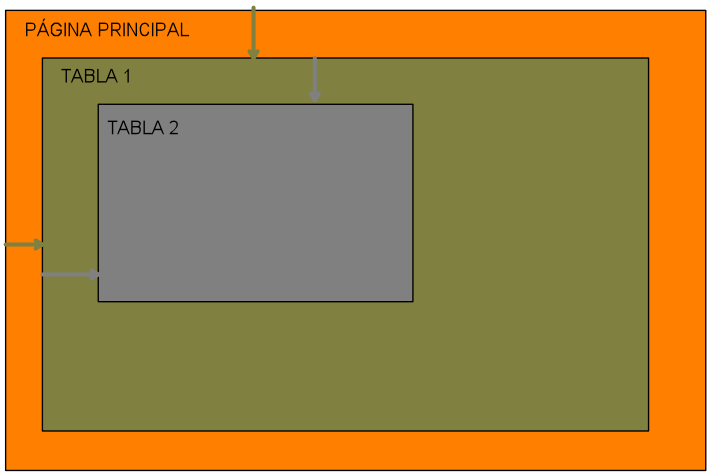


Ilustración 59 Tablas anidadas.

La solución es bastante sencilla, mientras que un elemento tenga padre, le sumamos los desplazamientos del padre de forma recursiva. Esto se ha explicado detenidamente para la mejor comprensión del código del ejemplo cuyo resultado se muestra en la figura siguiente.

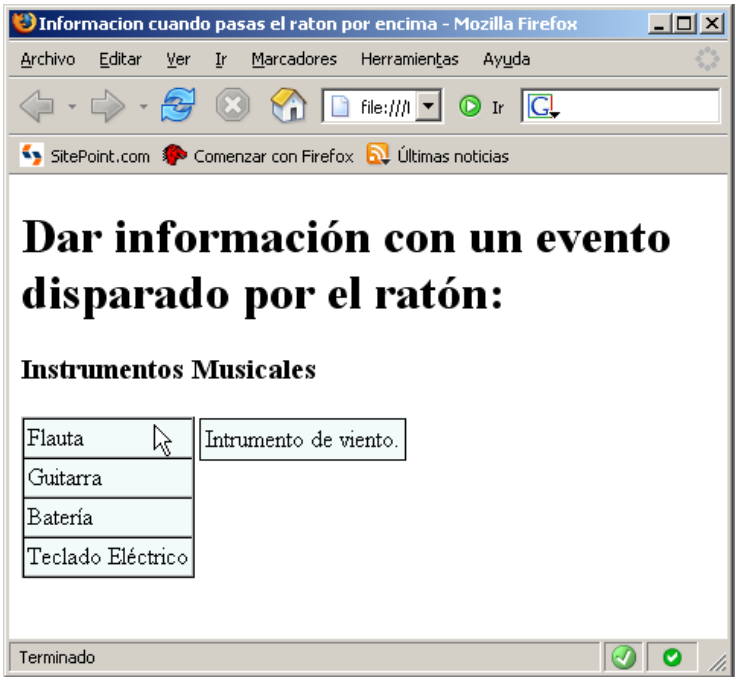


Ilustración 60 Generación de cuadros de información dinámicos.

Con todo lo anterior explicado ya, el ejemplo consta de solo un archivo del cual dividiremos el código en parte HTML y parte JavaScript.

Informacion1.html(Parte JavaScript)

```
function cogerInfo(Elemento)
{
  crearInformacion(Elemento);
  posicionVentanaInformacion(Elemento);
}
```

```
function borrarInfo()
{
  var ventanaborrar = document.getElementById("ventanaDatosCuerpo");
  var indice = ventanaborrar.childNodes.length;
    for (var i = indice - 1; i >= 0 ; i--)
    {
    ventanaborrar.removeChild(ventanaborrar.childNodes[i]);
    }
  document.getElementById("ventanaInformacion").style.border = "none";
}

function crearInformacion(Elemento)
{
//1.Creamos la fila
var fila = document.createElement("tr");
//2.Creamos la columna con la información.
var celda1 = document.createElement("td");
      switch(Elemento.id)
      {
      case "instrumento0": celda1.innerHTML = "Intrumento de viento.";
      break;
      case "instrumento1": celda1.innerHTML = "Instrumento de
cuerda.";
      break;
      case "instrumento2": celda1.innerHTML = "Instrumento de
percusión.";
      break;
      case "instrumento3": celda1.innerHTML = "Instrumento
Electrónico.";
      break;
      }
//3.Añadimos la columna a la fila y la fila a la tabla.
fila.appendChild(celda1);
document.getElementById("ventanaDatosCuerpo").appendChild(fila);
}

function posicionVentanaInformacion(Elemento)
{
//1.Vamos a reposicionar la ventana de información respecto al
elemento que ha pedido información.
var reposicionar = document.getElementById("ventanaInformacion");
//2.Calcular la posición absoluta que ocupara la ventana.
/* 2.1 Para calcular el desplazamiento a la izquierda cogemos el ancho
de la tabla de al lado que es casi todo el desplazamiento, luego le
sumamos algo. */
var izquierdaDesplazado =  Elemento.offsetWidth + 13;
/* 2.2 Para calcular lo desplazado que esta el elemento del borde
superior usamos una función, le pasamos el elemento y le decimos el
borde que queremos calcular. */
var altoDesplazado = calcularEsquina(Elemento, "offsetTop");
//3.Le aplicamos las propiedades calculadas.
reposicionar.style.border = "black 1px solid";
reposicionar.style.left = izquierdaDesplazado + "px";
reposicionar.style.top = altoDesplazado + "px";
}

function calcularEsquina(elemento, atributoEsquina)  //ej: campo =
objetoDeLaPagina , atributoEsquina = "offsetTop"
{
var desplazamiento = 0;
  while(elemento) //Mientras exista el elemento
  {
  desplazamiento += elemento[atributoEsquina]; //Le sumamos al
  desplazamiento, el desplazamiento del elemento.
  /*Normalmente cada elemento solo contiene su desplazamiento respecto
  al padre, no de manera absoluta, así que pasamos al elemento padre,
  si existe en la siguiente iteración se sumara su desplazamiento
  también, sino terminara. */
  elemento = elemento.offsetParent;
  }
```

```
return desplazamiento;
}
```

Informacion1.html(Parte HTML)

```
<body>
<h1>Dar información con un evento disparado por el ratón:</h1>
<h3>Instrumentos Musicales</h3>

<table id="instrumentos" bgcolor="#F2FAFA" border="1" cellspacing="0"
cellpadding="2">
<tbody>
  <tr>
    <td id="instrumento0" onmouseover="cogerInfo(this);"
    onmouseout="borrarInfo();">Flauta</td>
  </tr>
  <tr>
    <td id="instrumento1" onmouseover="cogerInfo(this);"
    onmouseout="borrarInfo();">Guitarra</td>
  </tr>
  <tr>
    <td id="instrumento2" onmouseover="cogerInfo(this);"
    onmouseout="borrarInfo();">Batería</td>
  </tr>
  <tr>
    <td id="instrumento3" onmouseover="cogerInfo(this);"
    onmouseout="borrarInfo();">Teclado Eléctrico</td>
  </tr>
</tbody>
</table>

<div style="position:absolute;" id="ventanaInformacion">
  <table id="ventanaDatos" bgcolor="#F2FAFA">
    <tbody id="ventanaDatosCuerpo"></tbody>
  </table>
</div>

</body>
```

Como se ve en una de las líneas subrayadas de la parte HTML se puede asignar más de un evento a una etiqueta, en este caso se hace desaparecer el elemento si quitamos el ratón de encima debido a que solo estamos dando información, no es un menú, la parte más importante de este ejemplo es cómo calculamos la posición y generamos la tabla dinámica.

4.9.2 Ejemplo 2 – Tabla relativa al puntero del ratón.

Como se ha mostrado en la figura 4.11, para dar información al usuario típicamente se emplea una tabla adyacente a otra o se muestra la información junto al puntero del ratón en un cuadro. Este segundo ejemplo encara precisamente esta segunda vertiente, que es muchísimo mas sencilla ya que solo tenemos que coger las coordenadas del puntero e incrustar la tabla en una posición adyacente.

El aspecto gráfico del siguiente ejemplo se encuentra en la imagen siguiente:

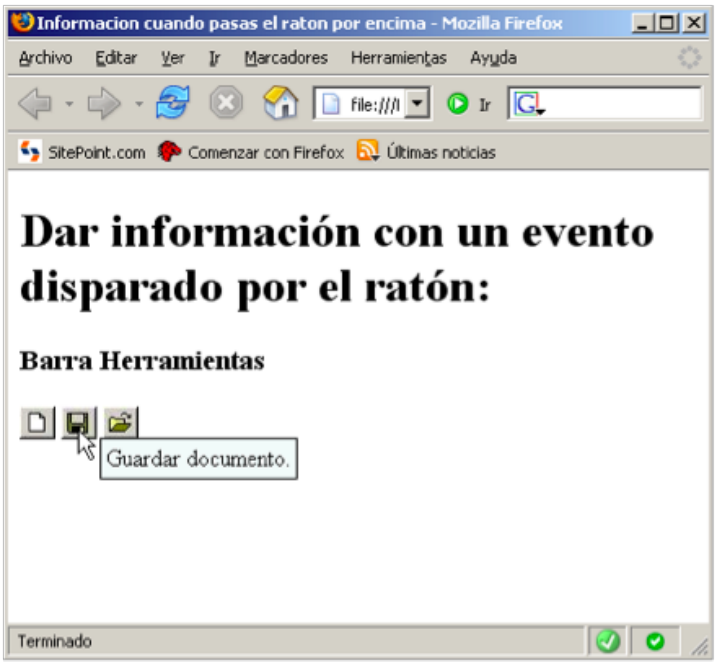


Ilustración 61 Tabla colocada donde se encuentra el puntero del ratón.

Si nos fijamos bien, no es que coloquemos la tabla encima justo del ratón, esto quedaría un poco antiestético, normalmente se coloca debajo de él o un poco a la derecha como es el caso, sumándole unos pocos píxeles a las coordenadas. El ejemplo consta de un solo archivo que es el siguiente, divido también en parte HTML y JavaScript

Informacion2.html (Parte HTML)

```
<body>
<h1>Dar información con un evento disparado por el ratón:</h1>
<h3>Barra Herramientas</h3>

<img src="0.jpg" id="Nuevo documento."
onmouseover="cogerInfo(this,event);" onmouseout="borrarInfo();"  />
<img src="1.jpg" id="Guardar documento."
onmouseover="cogerInfo(this,event);" onmouseout="borrarInfo();" />
<img src="2.jpg" id="Abrir documento."
onmouseover="cogerInfo(this,event);" onmouseout="borrarInfo();" />

<div style="position:absolute;" id="ventanaInformacion">
  <table id="ventanaDatos" bgcolor="#F2FAFA">
  <tbody id="ventanaDatosCuerpo"></tbody>
  </table>
</div>
</body>
```

Informacion2.html (Parte JavaScript)

```
function cogerInfo(Elemento, event) {
  crearInformacion(Elemento);
  posicionVentanaInformacion(Elemento, event);
}

function borrarInfo() {
  var ventanaborrar = document.getElementById("ventanaDatosCuerpo");
  var indice = ventanaborrar.childNodes.length;
    for (var i = indice - 1; i >= 0 ; i--)
    {
      ventanaborrar.removeChild(ventanaborrar.childNodes[i]);
    }
  document.getElementById("ventanaInformacion").style.border = "none";
```

```
}

function crearInformacion(Elemento){
  //1.Creamos la fila
  var fila = document.createElement("tr");
  //2.Creamos la columna con la información.
  var celda1 = document.createElement("td");
  celda1.innerHTML = Elemento.id;
  //3.Añadimos la columna a la fila y la fila a la tabla.
  fila.appendChild(celda1);
  document.getElementById("ventanaDatosCuerpo").appendChild(fila);
}

function posicionVentanaInformacion(Elemento,event) {
  1. Vamos a reposicionar la ventana de información respecto al
  elemento que ha pedido información.
  var reposicionar = document.getElementById("ventanaInformacion");
  /* 2.El objeto event, existe solo en el momento que ocurre un evento
  por si lo necesitas en el manejador y contiene cosas como la
  posición y estado de las teclas y ratón. */
  var ancho = event.clientX;
  var alto = event.clientY;
  //3.Le aplicamos las propiedades calculadas.
  reposicionar.style.border = "black 1px solid";
  reposicionar.style.left = ancho + 15 + "px";
  reposicionar.style.top = alto + "px";
}
```

El lector podrá apreciar que se ha dedicado mucho tiempo en el apartado 4.9 en explicar algo que no afecta directamente a AJAX (ya que no implica un tráfico de información dinámica con el servidor) sino que es más bien un artificio gráfico que juega con coordenadas, esto es porque este artificio gráfico se va a usar junto con AJAX y es en parte el culpable de que se puedan hacer cosas más que interesantes.

4.10 Auto completado empleando AJAX.

Para explicarlo rápidamente lo mejor es ver el resultado primero, mostrado en la siguiente ilustración.

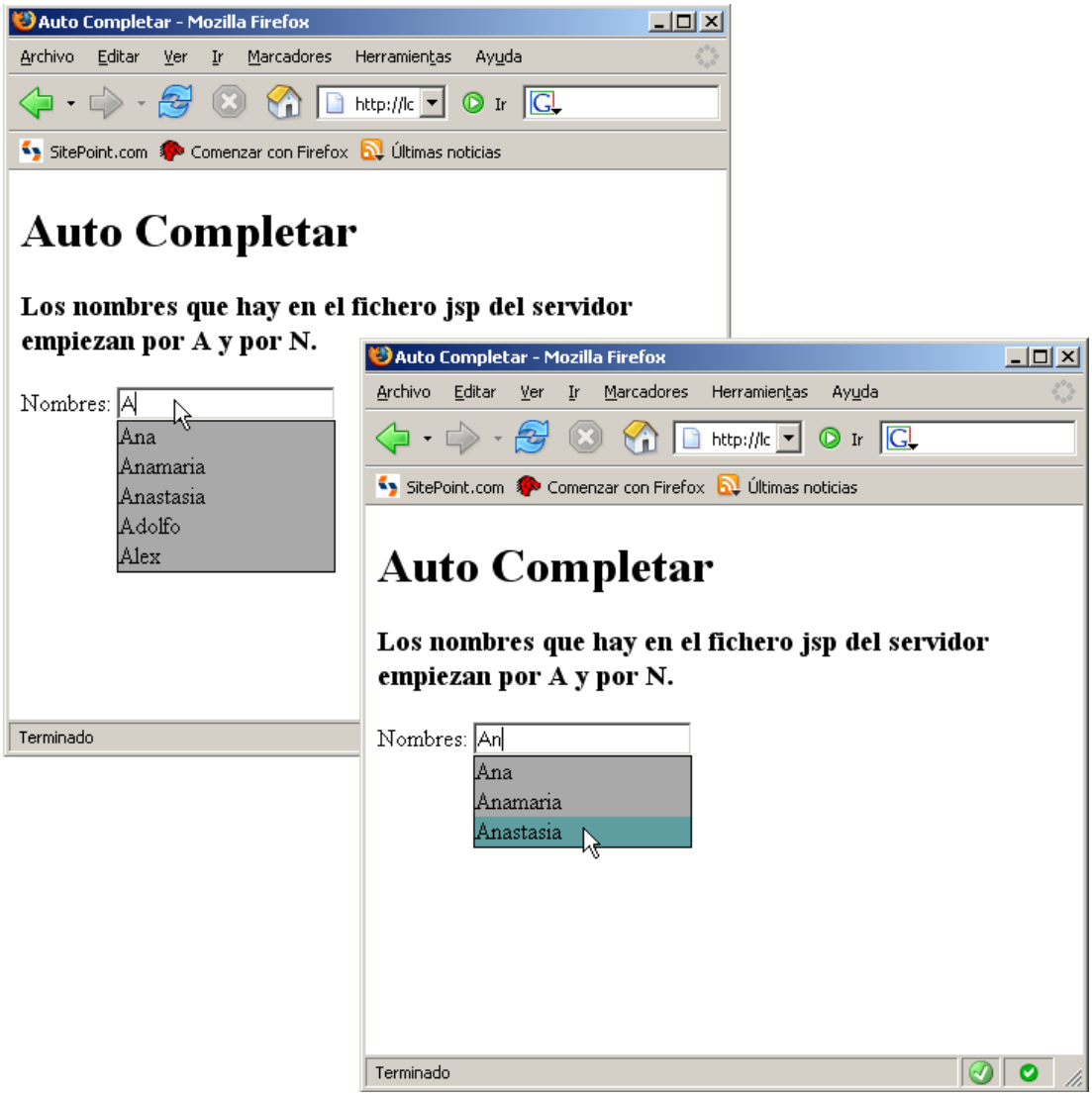


Ilustración 62 Ejemplo de auto completado empleando AJAX.

Como el lector puede ver en la ilustración no existe un botón de aceptar la selección, tenemos que elegir de entre las que hay y entonces se lanzaría la petición, aunque en una hipotética búsqueda en una base de datos estamos ahorrando tiempo y esto es verdad, a cambio cada vez que pulsamos una tecla se está lanzando una petición al servidor con toda la sobrecarga que ello supone.

Esto mismo lo podemos ver hecho por Google en Google Suggest mostrado en la figura que sigue.

Ilustración 63 Google suggest.

Si el lector recuerda las operaciones con cuadritos del apartado 4.9, este mismo tipo de cálculos se realizan para colocar dinámicamente la tabla justo debajo del recuadro de escritura, dando la sensación de ser un menú que se despliega automáticamente.

El ejemplo que nos ocupa tiene dos archivos sin contar las librerías AJAX, "autoCompletar.html" y "pedirInfo.jsp", como el archivo "autoCompletar.html" no es excesivamente largo lo pondremos completo sin cortes, contiene tres partes: una pequeña hoja de estilo, el código Javascript y el código HTML, mientras que a la página jsp simplemente le llega la cadena de texto que hay en el cuadro y devuelve un documento XML con las referencias encontradas.

autoCompletar.html

```
<html>
<head>
<title>Auto Completar</title>
<style type="text/css">
  .ratonFuera
  {
  background: #A9A9A9;
  color: #000000;
  }

  .ratonEncima
  {
  background: #5F9EA0;
  color: #000000;
  }
</style>
<script language="JavaScript" type="text/javascript"
src="lib/ConstructorXMLHttpRequest.js"></script>
<script language="JavaScript" type="text/javascript"
src="lib/ClasePeticionAjax.js"></script>
<script language="JavaScript" type="text/javascript">

function buscarNombres(elemento)
{
  if(elemento.value == "") //Si no hay nada escrito es que el usuario
  ha borrado, así que borramos la lista.
  {
  borrarInfo();
  return;
  }

var elementoActual = elemento; //Necesitaremos la variable en una
función interior, así que la redeclaramos.
```

```
borrarInfo(); //Borramos la información de la ventana, por si quedaba
algo, no se añada al final lo nuevo y se repita.
pedirInfo01= new objetoAjax("GET"); //Construimos un objetoAjax que
utilizará el método GET

//Cuando se usa el método GET la página a la que enviamos los datos y
los datos van unidos en la misma cadena.
var cadenaPeticionGET = "pedirInfo.jsp?"; //La página.
var datos ="&cadenaTexto=" + elemento.value; //Los datos
cadenaPeticionGET =cadenaPeticionGET + datos; //Unimos la página con
el resto de los datos.

/*Asignamos la función de completado, que llevara la carga de generar
las nuevas celdas con la información recibida, en esta ocasión,
aunque lo mas correcto sería que tuviéramos una función propia que lo
haga, esto es solo un ejemplo sencillo */
pedirInfo01.completado =
  function objetoRequestCompletado01(estado, estadoTexto,
  respuestaTexto, respuestaXML)
  {
  //Cogemos lo que nos interesa de XML.
  var documentoCompleto = respuestaXML.getElementsByTagName("nombre");
     if(documentoCompleto.length != 0) //Si tenemos alguna respuesta
     añadimos información, sino no.
     {
             posicionVentanaInformacion(elementoActual);
        for(var i = 0; i < documentoCompleto.length; i++)
        {
          //1.Creamos la fila de la tabla y le damos la información.
            var fila = document.createElement("tr");
            var celda = document.createElement("td");
            .innerHTML = documentoCompleto[i].childNodes[0].nodeValue;
          //2.Le aplicamos el estilo para que sea interactivo
          dependiendo de los eventos.
            celda.onmouseout = function()
            {this.className='ratonFuera';}; // Referencia a la hoja CSS
            celda.onmouseover = function()
            {this.className='ratonEncima';}; // Referencia a la hoja
            CSS
            celda.setAttribute("bgcolor", "#A9A9A9");
          //3.Si pulsamos sobre uno de los sugeridos se sustituirá el
          valor de búsqueda por este.
            celda.onclick = function() { ponerNombre(this);};
          //4.Unimos la nueva fila a la tabla.
            fila.appendChild(celda);
            document.getElementById("cuerpoTablaSugerancias").appendChi
            ld(fila);
        }
     }
  }

pedirInfo01.coger(cadenaPeticionGET); //Enviamos tanto la página como
los datos en la misma cadena.
}

function borrarInfo()
{
var ventanaborrar =
document.getElementById("cuerpoTablaSugerancias");
var indice = ventanaborrar.childNodes.length;
  for (var i = indice - 1; i >= 0 ; i--)
  {
  ventanaborrar.removeChild(ventanaborrar.childNodes[i]);
  }
document.getElementById("tablaSugerancias").style.border = "none";
}

function posicionVentanaInformacion(elemento)
{
  //1.Vamos a reposicionar la ventana de información respecto al
```

```
  elemento que ha pedido información.
    var reposicionar = document.getElementById("tablaSugerancias");
  //2.Calcular la posición absoluta que ocupara la ventana.
    var anchoCelda = elemento.offsetWidth; //El mismo ancho que lel
    input.
    var izquierdaDesplazado = calcularEsquina(elemento, "offsetLeft");
    //El desplazamiento a la izquierda total del input
    //Desplazamiento del alto del input + el alto del input.
    var altoDesplazado = calcularEsquina(elemento, "offsetTop") +
    elemento.offsetHeight;
  //3.Le aplicamos las propiedades a la ventana.
    reposicionar.style.border = "black 1px solid";
    reposicionar.style.width = anchoCelda + "px";
    reposicionar.style.left = izquierdaDesplazado + "px";
    reposicionar.style.top = altoDesplazado + "px";
}

function calcularEsquina(elemento, atributoEsquina)  //ej: campo =
objetoDeLaPagina , atributoEsquina = "offsetTop"
{
var desplazamiento = 0;
  while(elemento) //Mientras exista el elemento
  {
  desplazamiento += elemento[atributoEsquina]; //Le sumamos al
  desplazamiento, el desplazamiento del elemento.
  /*Normalmente cada elemento solo contiene su desplazamiento respecto
  al padre, no de manera absoluta, así que pasamos al elemento padre,
  si existe en la siguiente iteración se sumara su desplazamiento
  también, sino terminara. */
  elemento = elemento.offsetParent;
  }
return desplazamiento;
}

function ponerNombre(celda)
{
  document.getElementById("nombres").value =
  celda.firstChild.nodeValue;
  borrarInfo();
}
</script>
</head>
<body>

<h1>Auto Completar</h1>
<h3>Los nombres que hay en el fichero jsp del servidor
empiezan por A y por N.</h3>

<form action="null">
Nombres: <input type="text" size="20" id="nombres"
onkeyup="buscarNombres(this);" style="height:20;"/>
</form>
<table id="tablaSugerancias" bgcolor="#A9A9A9" border="0"
cellspacing="0" cellpadding="0" style="position:absolute;">
<tbody id="cuerpoTablaSugerancias"></tbody>
</table>

</body>
</html>
```

Se han resaltado dos funciones para remarcar algo interesante:

- La función **buscarNombres** hace una petición bastante sencilla y comprensible: hace una petición en base a lo que el usuario ha escrito en el cuadro de texto.

- Cuando llega el resultado, éste va a la función **objetoRequestCompletado01** que recoge el documento XML, lo analiza sacando los nombres y crea la tabla dinámicamente

Todos los elementos que componen el ejemplo los hemos visto ya, y los hemos remarcado precisamente para dejar patente que ya se han cubierto todos los aspectos principales de la programación usando AJAX: a partir de aquí es la imaginación lo que más cuenta realmente.

Para terminar aquí está el archivo .jsp que realiza la búsqueda y genera el documento XML, se ha programado una búsqueda para que los lectores con pocos conocimientos del lenguaje puedan comprenderlo en vez de utilizar una función preconstruida.

pedirInfo.jsp

```
<%@ page import="java.io.*,java.util.*,java.lang.*"
contentType="text/xml; charset=UTF-8" pageEncoding="UTF-8"%>
<%
	response.setHeader("Cache-Control", "no-cache");
	//1.Recuperamos la cadena de la petición
	request.setCharacterEncoding("UTF-8");
	String prefijo = request.getParameter("cadenaTexto");
	//2.Creamos las sugerencias
	ArrayList listanombres = new ArrayList();
	listanombres.add("Ana");
	listanombres.add("Anamaria");
	listanombres.add("Anastasia");
	listanombres.add("Adolfo");
	listanombres.add("Alex");
	listanombres.add("Naomi");
	listanombres.add("Noelia");
	listanombres.add("Nora");
	listanombres.add("Nox");
	//3.Miramos las que coinciden y construimos un documento XML.
	//3.1 Variables necesarias
	Iterator i = listanombres.iterator();
	boolean valido = true;
	//3.2 Iteracion para construir el código (Es lineal y pesado).
	out.println("<listaNombres>");
	while( i.hasNext() ) {
		String nombre=(String)i.next();

		//Comprobación de igualdad de la vieja escuela.
		if (prefijo.length() <= nombre.length()) {
		char[] nombre2 = nombre.toCharArray();
		char[] prefijo2= prefijo.toCharArray();
		int j=0;
			while (j < prefijo.length() &&
					(prefijo2[j] == nombre2[j]) ) {
			j++;
			}
			valido = (j == prefijo.length());
		}
		else {
		valido = false;
		}

		if(valido) {
		out.println("<nombre>" + nombre + "</nombre>");
		valido = false;
		}
	}
	out.println("</listaNombres>");
	out.close();
%>
```

El ejemplo termina aquí; sólo comentar nuevamente que este tipo de pequeñas aplicaciones crea tantas peticiones al servidor como teclas pulsamos: si escribo un nombre de 6 caracteres se lanzaran 6 peticiones. Se podría poner una espera y que solo se hiciera la petición a los dos segundos de haberse modificado el texto, con lo que al usuario le daría tiempo de escribir varias letras y la búsqueda realizada por el servidor quedaría más refinada y eficiente.

Bibliografía

Aquí aparecen las referencias a libros y páginas web que se han empleado de referencia. Si algo se parece a lo expresado en este manual no es casualidad, ya que parte de los ejemplos aquí expuestos están basados en otros previamente estudiados. El objetivo ha sido utilizar aquéllos que ya han sido puestos a prueba y demostrado sus cualidades didácticas y de simplicidad, lo que abunda en un aprendizaje progresivo y fácil.

Libros

Christian Gross - Ajax Patterns and Best Practices - Ed. Apress, 2006
Ryan Asleson y Nathaniel T. Schutta - Foundations of Ajax - Ed. Apress, 2006

Pequeños o grandes tutoriales y donde buscarlos

Sang Shin 10-Week Free AJAX Programming
http://www.javapassion.com/ajaxcodecamp/
Sergio Gálvez JSP(Java Server Pages) www.sicuma.uma.es
Ángel Barbero Tutorial de XML www.lawebdelprogramador.com
Lola Cárdenas Curso de JavaScript www.lawebdelprogramador.com
José C. García Curso de Hojas de Estilo www.lawebdelprogramador.com

Webs

W3C www.w3schools.com
W3 http://www.w3.org/
Web estilo http://www.webestilo.com/html/