Contenido Procesado

1. petición: 52 veces
2. datos: 50 veces
3. objeto: 47 veces
4. servidor: 37 veces
5. respuesta: 34 veces
6. ajax: 32 veces
7. peticiones: 32 veces
8. api: 32 veces
9. tipo: 31 veces
10. función: 31 veces

DWEC04.- Programación ajax en JavaScript.

1.- Introducción a ajax.

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

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

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

- Conseguir una presentación basada en estándares, usando XHTML, CSS y un uso amplio de técnicas del DOM, para poder mostrar la información de forma dinámica e interactiva.
- Intercambio y manipulación de datos a través de ficheros JSON
- Uso de JavaScript, para unir todos los componentes.

Las tecnologías que forman ajax son:

- XHTML y CSS, para la presentación basada en estándares.
- DOM, para la interacción y manipulación dinámica de la presentación.
- XML, XSLT y JSON, para el intercambio y manipulación de información.
- `XMLHTTPRequest`, para el intercambio asíncrono de información.

- JavaScript, para unir todos los componentes anteriores.

El **modelo clásico de aplicaciones Web (desuso)** funciona de la siguiente forma:

- la mayoría de las acciones del usuario se producen en la interfaz, disparando solicitudes HTTP al servidor web. El servidor efectúa un proceso (recopila información, realiza las acciones oportunas), y devuelve los datos solicitados por el cliente en formato HTML.
- Este es un modelo adaptado del uso original de la Web como medio hipertextual, pero a nivel de aplicaciones de software, este tipo de modelo **no es necesariamente el más recomendable**. Y en la actualidad, como veremos más adelante, estas operaciones ya no se realizan de esta manera.
- Cada vez que se realiza una petición al servidor, el usuario lo único que puede hacer es esperar, ya que muchas veces la página cambia a otra diferente, y hasta que no reciba todos los datos del servidor, no se mostrará el resultado, con lo que el usuario no podrá interactuar de ninguna manera con el navegador.

ajax es la tecnología que consiguió solucionar estas limitaciones. A grosso modo, lo que se intenta evitar, son esencialmente esas esperas. El cliente podrá hacer solicitudes al servidor, mientras el navegador sigue mostrando la misma página web, y cuando el navegador reciba una respuesta del servidor, la mostrará al cliente y todo ello sin recargar o cambiar de página.

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

Son muchas las razones para usar ajax:

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

1.1.- Requerimientos previos.

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

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

Para la programación con ajax vamos a necesitar de un servidor web, ya que las peticiones

ajax que hagamos, las haremos a un servidor. Los componentes que necesitamos son:

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

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

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

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

servidor XAMPP (Apache, MySQL, PHP).

En cuanto a este módulo, como hemos venido haciendo hasta ahora, nos centraremos en el lado del cliente, y como siempre, siguiendo la arquitectura cliente-servidor (Front y Backend).

Por lo que para nuestras consultas, en principio, usaremos siempre servicios de terceros a través de **APIs**

1.1.1.- Ventajas y desventajas de ajax

ajax aporta importantes ventajas al campo de la creación de las aplicaciones web. Entre ellas:

- **Carga de contenido remoto en segundo plano**.
- Aporta una **gran facilidad para aplicar paradigmas de interfaces de usuarios novedosas**, como páginas con scroll infinito o aplicaciones web de página única.
- **Facilidad y versatilidad** para comunicar con APIs de terceros para integrar sus servicios en nuestras aplicaciones.
- **Resolución de problemas reales** de manera sencilla.
- **Validación de formularios eficaz** al facilitar la detección de errores de manera temprana.
- **Mejor uso del ancho de banda del usuario**.
- En la programación **back-end**, saber que un servicio puede ser destino de peticiones ajax, hace que **los servicios se programen de forma más eficiente e independiente**, animando a la creación de **APIs**, que son la base de la programación basada en servicios. Este tipo de paradigma (conocido como **SaaS**) ha impulsado un mayor desarrollo y eficiencia de las aplicaciones web.
- Facilita la **compatibilidad con servicores back-end de todo tipo de tecnologías**. El único requisito es que acepte peticiones **http** y que envíe los datos en formatos estándar como son **XML o JSON**.

Es importante saber también que ajax tiene sus desventajas. Las principales son:

- Dificultad, en bastantes casos, para que los buscadores indexen todos los contenidos que integramos en nuestras aplicaciones procedentes de peticiones ajax.
- Aunque los conceptos en el uso de ajax no son difíciles, la implementación sí tiene sus dificultades porque requiere un manejo avanzado de JavaScript en aspectos como funciones callback, promesas, programación asíncrona, programación basada en eventos, etc.
- Las peticiones de ajax siguen siendo visibles para los usuarios, por lo que son sensibles a ataques malintencionados.
- La barra de navegación no permite ir al estado anterior en la página. Las peticiones ajax no se reflejan en la navegación.
- Puede incrementar el tráfico hacia los servidores de Internet. El hecho de que podamos realizar peticiones http a servicios de Internet, puede facilitar la creación de aplicaciones que hagan peticiones http constantemente. Ante esto, muchos servidores usan protección anti **CORS**.

1.1.2.- CORS

CORS es el acrónimo de Cross-Origin Resource Sharing, Intercambio de Recursos de Origen cruzado. Las peticiones ajax son fáciles de automatizar, lo que permitiría realizar cientos o miles de peticiones con unas pocas líneas de código, lo que podrá provocar el colapso del servidor. Para que los servicios de internet se protejan y solo resuelvan peticiones procedentes de dominios validados, se ideó una norma que ahora es parte del protocolo http. Los datos sobre CORS se colocan en la cabecera de los paquetes http y permiten una vía de comunicación entre el navegador y el servidor web que asegura que se cumplen las normas establecidas.

Una política CORS habitual es que solo se resuelvan peticiones procedentes del dominio en el que está el servicio. Pero hay políticas más sofisticadas. Software de servidor web como **Apache** o **nginx** tienen capacidad para modificar esas políticas.

La cabecera http que protege a los servidores de vulnerabilidades causadas por CORS es **Access-Control-Allow-Origin**. Si a esta directiva se le asigna el valor * (asterisco), entonces admite cualquier origen para las peticiones. Se pueden especificar, en su lugar, dominios concretos y así solo se admiten peticiones de estos dominios (serán dominios confiables). Hay otras cabeceras que permiten matizar aún más estas políticas.

1.2.- Comunicación asíncrona.

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

- El usuario solicita algo al servidor.
- El servidor ejecuta los procesos solicitados (consulta a una base de datos, procesado de la información, etc.).
- Cuando el servidor termina, devuelve los resultados al cliente.

En el paso 2, mientras se ejecutan los procesos en el servidor, el cliente lo único que puede

hacer es esperar, ya que el navegador está bloqueado en espera de recibir la información con los resultados del servidor.

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

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

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

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

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

2.- api

Esta serie de protocolos son una parte fundamental en el funcionamiento de las aplicaciones y webs actuales. Seguramente has oído hablar más de una vez sobre ellas, cuando determinado servicio popular pone límites en su api o crean nuevas para extender su uso en otras aplicaciones.

Vamos a empezar explicándote de la forma más sencilla que podamos qué es exactamente una api y cuáles son sus principales usos. Luego, terminaremos con algunos ejemplos con los que vas a poder hacerte una idea de para qué sirven y cómo son utilizadas en la práctica.

Qué es una api

El término api es una abreviatura de *Application Programming Interfaces*, que en español significa *interfaz de programación de aplicaciones*. Se trata de un conjunto de definiciones y protocolos que se utiliza para desarrollar e integrar el software de las aplicaciones, permitiendo la comunicación entre dos aplicaciones de software a través de un conjunto de reglas.

Application Programming Interfacesinterfaz de programación de aplicaciones

Así pues, podemos hablar de una api como una especificación formal que establece cómo un módulo de un software se comunica o interactúa con otro para cumplir una o muchas funciones. Todo dependiendo de las aplicaciones que las vayan a utilizar, y de los permisos que les dé el propietario de la api a los desarrolladores de terceros.

En otras palabras: Describe la manera apropiada para que un desarrollador de software componga un programa en un servidor que se comunica con varias aplicaciones cliente.

2.1.- api Rest

¿Qué es la api REST?

«REST»: *REpresentational State Transfer*, es un tipo de arquitectura de desarrollo web que se apoya totalmente en el estándar HTTP. Fue definida en el 2000 por Roy Fielding, uno de los padres de la especificación HTTP y un referente en la arquitectura de redes. Fielding definió REST como un estilo arquitectónico y una metodología de uso frecuente en el desarrollo de servicios de Internet, como los sistemas hipermedia distribuidos:

REpresentational State Transfer

La forma completa de REST api es la interfaz de programación de aplicaciones de transferencia de estado representacional, más comúnmente conocida como servicio web REST api. Significa que cuando se llama a una api RESTful, el servidor transferir a representación de los recursos solicitados estado al sistema del cliente.

La forma completa de REST api es la interfaz de programación de aplicaciones de transferencia de estado representacional, más comúnmente conocida como servicio web REST api. Significa que cuando se llama a una api RESTful, el servidor transferir a representación de los recursos solicitados estado al sistema del cliente.

Por ejemplo, cuando un desarrollador realiza una solicitud a la api de Twitter para recuperar la información de la pagina principal de este, la api devolverá el estado de ese usuario, su nombre, seguidores y publicaciones compartidas en Twitter.

Es decir, **la api es una capa de abstracción que ejerce intermediario entre el cliente y el servidor,** a la que los desarrolladores realizan consultas y esta devuelve los datos que se le han solicitado, o realiza los procesos que se le han pedido.

Hoy por hoy, la mayoría de las aplicaciones que se desarrollan para servicios profesionales disponen de una api REST para el intercambio de información entre el front y el back. Lo que la hace tan potente es precisamente el aislamiento que proporciona entre la lógica del back-end y cualquier cliente consumidor de éste. Esto le permite ser usada por cualquier tipo de cliente: web, móvil, etc. Así, cualquier dispositivo/cliente que entienda de HTTP puede hacer uso de su propia api REST de manera muy simple. Esto ha hecho que en los últimos años este tipo de arquitectura haya ganado peso frente a otras más complejas como SOAP, para el intercambio y manipulación de datos.

2.1.1 - Usos de la especificación HTTP

Para el desarrollo de una api REST es necesario un conocimiento profundo de la

especificación HTTP, sobre todo en lo referente a métodos permitidos, códigos de estado y aceptación de tipos de contenido.

En este apartado vamos a explorar los principales métodos que tiene una api Rest y que nosotros utilizaremos cómo desarrolladores, y los códigos que nos puede devolver esta en función de la consulta que realicemos.

Los métodos son usados para manipular los diferentes recursos que conforman la api. Los principales métodos soportados por HTTP y por ello usados por una api REST son:

- POST: crear un recurso nuevo.
- PUT: modificar un recurso existente.
- GET: consultar información de un recurso.
- DELETE: eliminar un recurso determinado.
- PATCH: modificar solamente un atributo de un recurso.

Estos métodos junto con la URI, nos proporciona una interfaz uniforme que nos permite la transferencia de datos en el sistema REST aplicando operaciones concretas sobre un recurso determinado. Aunque la mayoría de las operaciones que componen una api REST podrían llevarse a cabo mediante métodos GET y POST, el abuso de ellos para operaciones que nada tienen que ver con el propósito con el que se concibieron, puede provocar un mal uso del protocolo alejado del estándar o la construcción de URIs con nomenclatura errónea mediante el uso de verbos.

Cuando se realiza una petición determinada, es de vital importancia conocer si dicha operación se ha llevado a cabo de manera satisfactoria o por el contrario se ha producido algún tipo de error. Para ello, HTTP dispone de un amplio número de códigos de error/éxito que cubren todas las posibles respuestas que el usuario puede recibir cuando trata de manipular un recurso mediante el uso de una api REST.

Las más comunes son:

- 200 OK. respuesta estándar para peticiones correctas.
- 201 Created. La petición ha sido completada y ha resultado en la creación de un nuevo recurso.
- 202 Accepted. La petición ha sido aceptada para procesamiento, pero este no ha sido completado.
- 400 Bad Request. La solicitud contiene sintaxis errónea.
- 403 Forbidden. La solicitud fue legal, pero el servidor rehúsa responder dado que el cliente no tiene los privilegios para hacerla.
- 404 Not Found. Recurso no encontrado. Se utiliza cuando el servidor web no encuentra la página o recurso solicitado.
- 500 Internal Server Error. Es un código comúnmente emitido por aplicaciones empotradas en servidores web, cuando se encuentran con situaciones de error ajenas a la naturaleza del servidor web.

2.2. - Realización de consultas a APIs

Como hemos comentado anteriormente, el concepto de api Rest se empezó a emplear en el año 2000, por lo que a lo largo de la evolución de este concepto y más en la ultima década, con el auge de las tecnologías multiplataforma y los servicios web, ha cambiado mucho la forma en la que se utilizan este tipo de recursos.

Vamos a explorar las diferentes tecnologías que se han usado a lo largo de los últimos años para utilizar las APIs:

- HttpRequest
- Fetch
- JQuery

2.2.1.- La api XMLHttpRequest.

Aunque actualmente el uso de `XMLHTTPRequest` **esté cada vez más en desuso**, es importante tener una pincelada de los orígenes de la tecnología, para entender qué es lo que se usa actualmente y porque.

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

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

El concepto que está detrás del objeto `XMLHTTPRequest`, surgió gracias a los desarrolladores de Outlook Web Access (de Microsoft), en su desarrollo de Microsoft Exchange Server 2000. La interfaz `IXMLHTTPRequest`, se desarrolló e implementó en la segunda versión de la librería MSXML, empleando este concepto. Con el navegador Internet Explorer 5.0 en Marzo de 1999, se permitió el acceso a dicha interfaz a través de ActiveX.

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

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

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

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

En febrero de 2008, la W3C publicó otro borrador denominado "`XMLHTTPRequest` Nivel 2". Este nivel consiste en extender la funcionalidad del objeto `XMLHTTPRequest`, incluyendo, pero no limitando, el soporte para peticiones **cross-site**, gestión de **byte streams**, progreso de eventos, etc. Esta última revisión de la especificación, sigue estando en estado "working draft" (borrador), a septiembre de 2010.

2.2.1.1.- Creación del objeto XMLHttpRequest.

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

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

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

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

```
                }
                catch (errorControlado) {}//Capturamos el error,
            }

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

     Más información sobre gestión de errores en:
     HTTP://www.javascriptkit.com/javatutors/trycatch2.sHTML */
     throw new Error("No se pudo crear el objeto XMLHTTPRequest");
}
// para crear un objeto XHR lo podremos hacer con la siguiente llamada.
var objetoAJAX = new objetoXHR();
```

2.2.1.2.- Métodos del objeto XMLHttpRequest.

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

Metodo	Descripción
`abort()`	Cancela la solicitud actual.
`getAllResponseHeaders()`	Devuelve la información completa de la cabecera.
`getResponseHeader()`	Devuelve la información específica de la cabecera.
`open(metodo, url, async, usuario, password)`	Especifica el tipo de solicitud, la URL, si la solicitud se debe gestionar de forma asíncrona o no, y otros atributos opcionales de la solicitud. • método: indicamos el tipo de solicitud: `GET` o `POST`. • `url`: la dirección del fichero al que le enviamos las peticiones en el servidor. • `async`: true (asíncrona) o false (síncrona). • usuario y password: si fuese necesaria la autenticación en el

Especifica el tipo de solicitud, la URL, si la solicitud se debe gestionar de forma asíncrona o no, y otros atributos opcionales de la solicitud.

- método: indicamos el tipo de solicitud: `GET` o `POST`.
- `url`: la dirección del fichero al que le enviamos las peticiones en el servidor.
- `async`: true (asíncrona) o false (síncrona).
- usuario y password: si fuese necesaria la autenticación en el

`send(datos)`

- `send(string)` Envía la solicitud al servidor.
- datos: Se usa en el caso de que estemos utilizando el método `POST`, como método de envío. Si usamos `GET`, datos será `null`.

- `send(string)` Envía la solicitud al servidor.
- datos: Se usa en el caso de que estemos utilizando el método `POST`, como método de envío. Si usamos `GET`, datos será `null`.

`setRequestHeader()` Añade el par etiqueta/valor a la cabecera de datos que se enviará al servidor.

Para probar el siguiente código, que incluye una petición ajax, tienes que hacerlo a través del servidor web. Para ello debes extraer el ejemplo, dentro de la raíz del servidor web, arrancar el servidor web e ir a la dirección `HTTP://localhost/web/dwec07132`

Puedes utilizar Firebug, para comprobar como se realiza la petición ajax.

```
function cargarSync(objeto, url)
{
     if (miXHR)
     {
          alert("Comenzamos la petición ajax");
          //Si existe el objeto  miXHR
          miXHR.open('GET', url, false); //Abrimos la url, false=SINCRONA

          // Hacemos la petición al servidor. Como parámetro del método send:
          // null -> cuando usamos GET.
          // cadena con los datos -> cuando usamos POST
          miXHR.send(null);
          //Escribimos la respuesta recibida de la petición ajax en el objeto DIV
          textoDIV(objeto, miXHR.responseText);
          alert("Terminó la petición ajax");
     }
```

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

2.2.1.3.- Propiedades del objeto XMLHttpRequest.

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

Propiedad	Descripción
`onreadystatechange`	Almacena una función (o el nombre de una función), que será llamada automáticamente, cada vez que se produzca un cambio en la propiedad `readyState`.
`readyState`	Almacena el estado de la petición `XMLHTTPRequest`. Posibles estados, del 0 al 4: • 0: solicitud no inicializada. • 1: conexión establecida con el servidor. • 2: solicitud recibida. • 3: procesando solicitud. • 4: solicitud ya terminada y la respuesta está disponible.

- 0: solicitud no inicializada.
- 1: conexión establecida con el servidor.
- 2: solicitud recibida.
- 3: procesando solicitud.
- 4: solicitud ya terminada y la respuesta está disponible.

`responseText`	Contiene los datos de respuesta, como una cadena de texto.
`responseXML`	Contiene los datos de respuesta, en formato XML.
`status`	Contiene el estado numérico, devuelto en la petición al servidor (por ejemplo: "404" para "No encontrado" o "200" para "`OK`").
`statusText`	Contiene el estado en formato texto, devuelto en la petición al servidor (por ejemplo: "`Not Found`" o "`OK`").

Para probar el siguiente código, que incluye una petición ajax, tienes que hacerlo a través del servidor web. Para ello debes extraer el ejemplo dentro de la raíz del servidor web, arrancar el servidor web e ir a la dirección `HTTP://localhost/web/dwec07133`

Puedes utilizar Firebug, para comprobar como se está realizando la petición ajax.

```
function cargarAsync(objeto, url)

     {

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

          // Hacemos la petición al servidor. Como parámetro:
```

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

2.2.2.- Realizar peticiones mediante la api Fetch

Desde hace unos años, para realizar este tipo de consultas desde JavaScript nativo, se emplea una api llamada Fetch, que es compatible con el uso de promesas, lo que la permite aprovechar las nuevas capacidades de JavaScript a la par que adapta el uso de ajax a los servicios actuales e implementa un modelo de trabajo más sencillo y mantenible.

La interfaz de Fetch proporciona un método global que se llama precisamente **fetch**. Este método requiere, al menos, indicar la URL del destino de la petición. El resultado de fetch es una promesa, que se considera resuelta cuando se reciben resultados sin error del destino.

fetch(direccionServicio)

- **.then**(función que recibe el objeto respuesta si la petición finaliza bien)
- **.catch**(función que recibe el error producido durante la petición)

Ejemplo:

```
fetch( "https://alumno.net/servicios/nifaleatorio-php")

.then (response=> {

    console.log (response.status);

})

.catch (error=> {

    console.log ("Error: " + error);

});
```

Este código realiza una petición **http** de tipo GET a la dirección indicada, la cual proporciona

un servicio que devuelve números NIF calculados de forma aleatoria y que se pueden usar para hacer pruebas en bases de datos y otras aplicaciones.

Como **fetch** retorna una promesa, el método **then** permite procesar el resultado. El resultado de la promesa es un objeto de respuesta que se conoce como **response** que representa el paquete http que proporcionan los datos requeridos. En este caso, la propiedad status del objeto response devolvería 200, código http que significa **OK** y ese será el mensaje que veríamos en la consola.

El método catch recoge el error ocurrido si la petición no concluye bien. Observemos este código:

```
fetch ( "http://noexiste.com")

.then ( response=> {

    console.log (response.status);

})

.catch ( error=> {

    console.log ( "Error: " +error);

});
```

Este código retornará un código de error porque la petición no se puede resolver. Concretamente el texto que aparecería por consola sería:

Error: TypeError: Failed to fetch

Recomendación

Cabe destacar, que debido a las facilidades que nos proporciona hoy en día la librería JQuery, cada vez se tiende a realizar más este tipo de consultas a través de esta, aunque está perfectamente realizada una consulta usando promesas desde JavaScript nativo.

2.2.2.1.- Manipular la respuesta. objeto Response

Como hemos visto en el apartado anterior, cuando una petición http realizada con fetch finaliza correctamente, el método then recibe un objeto conocido como objeto de respuesta o response.

En la siguiente tabla, enumeraremos las propiedades y métodos de este objeto:

Propiedad o método	USO
headers	Obtiene un objeto que contiene las cabeceras http del paquete de respuesta
body	Cuerpo de la respuesta http.

status	Devuelve el código de respuesta de la petición http.
statusText	Devuelve un texto descriptivo del código de respuesta de la petición.
ok	Con valor true, indica que la petición se resolvió sin problemas.
redirected	Con valor true, indica que la respuesta es el resultado de una redirección.
url	Devuelve la URL de la que procede la respuesta.
type	Indica el tipo de respuesta de la petición. Posibles valores: • basic: respuesta coherente con la petición original. Muestra todas las cabeceras salvo las relacionadas con cookies. • cors: Indica que la respuesta procede de un origen CORS admitido. Hay cabeceras que no se pueden ver. • error: Hay error de red. • opaque: respuesta para una petición marcada como no-cors; es decir, sin usar CORS.

Indica el tipo de respuesta de la petición. Posibles valores:

- basic: respuesta coherente con la petición original. Muestra todas las cabeceras salvo las relacionadas con cookies.
- cors: Indica que la respuesta procede de un origen CORS admitido. Hay cabeceras que no se pueden ver.
- error: Hay error de red.
- opaque: respuesta para una petición marcada como no-cors; es decir, sin usar CORS.

redirect(URL)	Método que redirige la respuesta a otra URL.
clone()	Clona la respuesta en otro objeto.
error()	Clona la respuesta generando un error de red.
text()	Método que sirve para obtener un flujo de texto de la respuesta. Devuelve una promesa que, cumplida, resuelve obteniendo los datos de la respuesta en formato texto.

Método que sirve para obtener un flujo de texto de la respuesta.

Devuelve una promesa que, cumplida, resuelve obteniendo los datos de la respuesta en formato texto.

json() Método similar al anterior, pero que trata de convertir la respuesta en un objeto de tipo JSON.

Para ello crea una nueva promesa donde, si finaliza de forma correcta, la resolución es dicho objeto JSON.

Método similar al anterior, pero que trata de convertir la respuesta en un objeto de tipo JSON.

Para ello crea una nueva promesa donde, si finaliza de forma correcta, la resolución es dicho objeto JSON.

blob() Genera una promesa que es resuelta como un objeto binario en el caso de que finalice correctamente.

2.2.2.2.- Personalizar la petición. objeto Request

Cuando se realizan peticiones mediante el método fetch, además de la URL, podemos pasar como parámetro un objeto de tipo Request. Este objeto permite modificar el paquete http que se envía con la petición para que recoja aspectos que nos interesen y que la petición sea lo más ajustada posible a nuestros intereses.

Las propiedades de las que disponemos en los objetos request son:

PROPIEDAD	USO
url	Propiedad obligatoria que contien la URL destino de la petición
method	Método http que se usará en la petición. Normalmente se usa **GET** o **POST**, pero es válido cualquier comando http:
headers	Permite indicar un objeto de tipo Headers que permite modificar las cabeceras http de la petición.
mode	Indica si se usa CORS en la petición. Posibilidades: • **same-origin**. Solo se acepta respuesta del mismo dominio en el que se ha realizado la petición. • **cors**. Se permiten respuestas de dominios cruzados. • **no-cors**. Solo se admiten métodos GET, POST o PUT. • **navigate**. Indica que la petición se usa como URL del navegador y no para ser usada como parte de una petición ajax

Indica si se usa CORS en la petición. Posibilidades:

- **same-origin**. Solo se acepta respuesta del mismo dominio en el que se ha realizado la petición.
- **cors**. Se permiten respuestas de dominios cruzados.
- **no-cors**. Solo se admiten métodos GET, POST o PUT.
- **navigate**. Indica que la petición se usa como URL del navegador y no para ser usada como parte de una petición ajax

cache Indica el modo de caché de la respuesta. Los navegadores cachean las respuestas a peticiones previas para acelerar la navegación.

Esta propiedad permite calibrar cómo deseamos el cacheado. Posibilidades:

- **default**. Valor habitual.
- **no-store**. Obliga siempre a traer la respuesta del servidor.
- **reload**. Obliga a traer la respuesta del servidor y no la de la caché, pero la respuesta se guarda.
- **no-cache**.
- **force-cache**. Fuerza a usar la caché
- **only-if-cached**. Solo se admiten los datos si proceden de la caché.

Indica el modo de caché de la respuesta. Los navegadores cachean las respuestas a peticiones previas para acelerar la navegación.

Esta propiedad permite calibrar cómo deseamos el cacheado. Posibilidades:

- **default**. Valor habitual.
- **no-store**. Obliga siempre a traer la respuesta del servidor.
- **reload**. Obliga a traer la respuesta del servidor y no la de la caché, pero la respuesta se guarda.
- **no-cache**.
- **force-cache**. Fuerza a usar la caché
- **only-if-cached**. Solo se admiten los datos si proceden de la caché.

redirect Indica cómo se deben de procesar las respuestas en el caso de que procedan de redirecciones. Posibilidades:

- **follow**. Se siguen las redirecciones sin problemas.
- **error**. Si hay redirecciones, se retorna un error.

Indica cómo se deben de procesar las respuestas en el caso de que procedan de redirecciones. Posibilidades:

- **follow**. Se siguen las redirecciones sin problemas.
- **error**. Si hay redirecciones, se retorna un error.

credentials Permite especificar si se admiten cookies en la petición. Posibilidades:

- **omit**. Nunca se envían ni reciben cookies.
- **same-origin**. Es el valor por defecto.
- **include**. Siempre se admiten cookies.

Permite especificar si se admiten cookies en la petición. Posibilidades:

- **omit**. Nunca se envían ni reciben cookies.
- **same-origin**. Es el valor por defecto.
- **include**. Siempre se admiten cookies.

integrity Almacena una cadena de integridad creada con un algoritmo hash de criptografía (se admite sha256, sha384 y sha512),

para validar la integridad de la petición.

Almacena una cadena de integridad creada con un algoritmo hash de criptografía (se admite sha256, sha384 y sha512),

para validar la integridad de la petición.

Ejemplo de petición usando objeto de tipo request:

```
fetch ( "http://direccionnovalida.com/" , {
  method: "POST",
  mode; "cors",
  cache: "no-cache"
})...
```

2.2.2.3.- Enviar datos con la petición

En muchas ocasiones, los servidores de Internet que otorgan servicios, requieren o permiten enviar datos para determinar de una forma más ajustada los datos que debe devolver. De forma clásica, esos datos son pares nombre/valor, listas parámetros con valores concretos. Pero actualmente se admite enviar datos incluso en formato de texto JSON.

Los documentos HTML siempre han permitido el envío de este tipo de datos a servidores externos a través de los formularios. De manera clásica, los datos se envían usando los comandos http GET o POST. El método GET envía los datos en la URL y POST lo hace en el cuerpo del mensaje http. Esa diferencia puede hacer pensar que POST es más seguro al ocultar más los datos que envía, pero realmente no lo es. Los datos viajan de forma plana y solo si lo

ciframos estarán realmente protegidos.

Actualmente, los comandos http se asocian a verbos que requieren un tipo concreto de acción. Así **GET** se asocia a una petición de datos y **POST** a un envío de datos a un servidor. Otros comandos http son: **PUT** (modificación de datos), **DELETE** (borrado de datos), **PATCH** (modificación parcial de datos) o **HEAD** (petición de cabecera).

Evidentemente hay que conocer la api del servicio final para saber qué método usar, cómo enviar los datos y cómo debemos recibirlos, dependiendo de lo que el servicio es capaz de aceptar.

2.2.2.4.- Uso de await/async con fetch

Puesto que la api Fetch es compatible con promesas, es posible manejarla mediante la notación await/async del estándar ES2017. Esta notación, para muchos desarrolladores, es más legible y facilita mucho el mantenimiento del código. El ejemplo visto en el apartado anterior para enviar datos de un supuesto POST al servicio de pruebas tripicode, sería, en esta notación, de esta forma:

```
let data={
    title: "Mi mensaje",
    body: "Mensaje de prueba",
    userId:5
}
async function peticion() {
  try {
    const resp=await
        fetch (`https://jsonplaceholder.typicode.com/posts`, {
        method: `POST`,
        body: JSON.stringify(data) ,
        headers: {
            "Content-type": "application/json; charset=UTF-8"
        }
       });
    const json=await resp.json();
    console.log(json);
  }
  catch(err) {
```

```
    console.log(err);

peticion();
```

2.2.3.- peticiones usando JQuery

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

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

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

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

- jQuery.
- prototype.
- scriptaculous.
- mooTools.
- dojo, swfobject, chrome-frame, webfont, etc.

google.load

Hay muchísimas librerías que se pueden utilizar para programar ajax, dependiendo del lenguaje que utilicemos.

Visita la siguiente URL para ver un listado de librerías para JavaScript.

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

2.2.3.1.- Introducción a jQuery (parte I).

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

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

escribe menos y produce más

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

Página Oficial de descarga de la librería jQuery.

Para saber más

Documentación oficial de la librería jQuery.

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

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

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

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

Cargando la librería desde nuestro propio servidor:

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

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

versión de producciónla versión de desarrollo`document.getElementById()`*`document.getElementById()`*.

Esta función `$("selector")`, acepta como parámetro una cadena de texto, que será un selector CSS, pero también puede aceptar un segundo parámetro, que será el contexto en el cuál se va a

hacer la búsqueda del selector citado. Otro uso de la función, puede ser el de `$(function){..});` equivalente a la instrucción `$(document).ready (function() {...});` que nos permitirá detectar cuando el DOM está completamente cargado.

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

2.2.3.2.- Introducción a jQuery (parte II).

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

Ejemplo usando el método tradicional con JavaScript:

```
     … Aquí irán las cabeceras y clase .colorido
<script type="text/javascript" src="funciones.js"></script>
<script type="text/javascript">
////////////////////////////////////////////////////////////////
// Cuando el documento esté cargado completamente llamamos a la función iniciar().
crearEvento(window,"load",iniciar);
function iniciar()
     var tabla=document.getElementById("mitabla"); // Seleccionamos la tabla.
     var filas= tabla.getElementsByTagName("tr");  // Seleccionamos las filas de la tabla.
     for (var i=0; i<filas.length; i++)
          if (i%2==1)     // Es una fila impar
          {
               // Aplicamos la clase .colorido a esas filas.
               filas[i].setAttribute('class','colorido');
          }
</script>
</head>
<body>
     .. Aquí irá la tabla …
</body>
</HTML>
```

Ejemplo equivalente al anterior, programado usando la librería jQuery:

```
// También podríamos poner $(function) {….});
$(document).ready(function()     // Cuando el documento esté preparado se ejecuta esta función.
     // Seleccionamos las filas impares contenidas dentro de mitabla y le aplicamos la clase colorido.
     $("#mitabla tr:nth-child(even)").addClass("colorido");
});
```

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

2.2.3.3.- función $.ajax() en jQuery.

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

La sintaxis de uso es: `$.ajax(opciones)`

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

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

Por ejemplo:

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

Veamos algunas propiedades de la función `$.ajax()` de jQuery:

Nombre	tipo	Descripción
`url`	`String`	La URL a la que se le hace la petición ajax.
`type`	`String`	El método HTTP a utilizar para enviar los datos: `POST` o `GET`. Si se omite se usa `GET` por defecto.

`data`	`Object`	Un objeto en el que se especifican parámetros que se enviarán en la solicitud. Si es de tipo `GET`, los parámetros irán en la URL. Si es `POST`, los datos se enviarán en las cabeceras. Es muy útil usar la función `serialize()`, para construir la cadena de datos.
`dataType`	`String`	Indica el tipo de datos que se espera que se devuelvan en la respuesta: XML, HTML, JSON, JSON`p, script, text` (valor por defecto en el caso de omitir `dataType`).
`success`	`Function`	función que será llamada, si la respuesta a la solicitud terminó con éxito.
`error`	`Function`	función que será llamada, si la respuesta a la solicitud devolvió algún tipo de error.
`complete`	`Function`	función que será llamada, cuando la solicitud fue completada.

Todos los parámetros de uso de la función $.ajax() de jQuery.

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

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

El método `.load()`

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

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

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

`load()`.

Ejemplos:

```
$("#noticias").load("feeds.HTML");

// carga en el contenedor con id noticias lo que devuelve la página feeds.HTML.

$("#objectID").load("test.php", { 'personas[]': ["Juan", "Susana"] } );

// Pasa un array de datos al servidor con el nombre de dos personas.
```

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

Debes conocer

Más información sobre el método .load() en jQuery.

La función `$.post()`

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

```
$.post("test.php");

$.post("test.php", { nombre: "Juana", hora: "11am" } );

$.post("test.php", function(resultados) {

     alert("datos Cargados: " + resultados);
```

Más información sobre el método .post() en jQuery.

La función `$.get()` y `$.getJSON()`

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

Más información sobre el método .get() en jQuery.

2.2.3.5.- Herramientas adicionales en programación ajax.

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

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

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

2.2.3.6.- Plugins jQuery.

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

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

Efectos con jQuery.

Documentación oficial de jQuery.

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

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

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

Desde la web oficial de jquery.com, puedes hojear todos los plugins disponibles para jQuery:

Plugins para jQuery.

Anexo.- Licencia de recursos