5. JavaScript y el navegador

5.1. AJAX

Uno de los usos más comunes en *JavaScript* es AJAX (*Asynchronous JavaScript And XML*), técnica (que no lenguaje) que permite realizar peticiones HTTP al servidor desde *JavaScript*, y recibir la respuesta sin recargar la página ni cambiar a otra página distinta. La información que se recibe se inserta mediante el API DOM que vimos en la sesión anterior.

Al igual que al abrir ventanas, el navegador restringe las peticiones al servidor para que sólo se puedan realizar sobre el mismo dominio al que pertenece la página que realiza la petición. Es decir, una página que desarrollamos en *localhost* no puede hacer una petición a *Twitter*.

Para usar AJAX, hemos de emplear el objeto `XMLHttpRequest` para lanzar una petición HTTP al servidor con el método `open(getPost, recurso, esAsync)` donde:

- `getPost`: cadena con el valor `GET` o `POST` dependiendo del protocolo deseado
- `recurso`: URI del recurso que se solicita
- `esAsync`: booleano donde `true` indica que la petición en asíncrona

Posteriormente se recibe la respuesta mediante la propiedad `responseText`.

Aunque en sus inicios era original de IE, posteriormente fue adoptado por todos los navegadores.

Vamos a dejar de lado IE8 y anteriores que en vez de usar el objeto `XMLHttpRequest` necesitan crear un objeto `ActiveXObject("Microsoft.XMLHTTP")`. Para trabajar con AJAX en versiones de navegadores antiguos se recomienda hacerlo mediante *jQuery*.

AJAX síncrono

Para comenzar vamos a basarnos en un ejemplo con llamadas síncronas (sí, parece extraño, es como si usáramos *SJAX*), ya que el código es un poco más sencillo. El siguiente fragmento realiza una petición a un archivo del servidor `fichero.txt`, y tras recibirlo, lo muestra mediante un diálogo de alerta.

```
var xhr = new XMLHttpRequest();
xhr.open("GET","fichero.txt", false);  ❶
xhr.send(null);  ❷
alert(xhr.responseText);  ❸
```

❶ Tras crear el objeto, se crea una conexión donde le indicamos el método de acceso (`GET` o `POST`), el nombre el recurso al que accedemos (`fichero.txt`) y finalmente si la llamada es asíncrona (`true`) o síncrona (`false`).
❷ Se envía la petición, en este caso sin parámetros (`null`)

❸ Recuperamos la respuesta con la propiedad `responseText`. Si el contenido del archivo hubiese estado en formato *XML*, usaríamos la propiedad `responseXML`

A modo de esquema, el siguiente gráfico representa las llamadas realizadas mediante una petición síncrona:

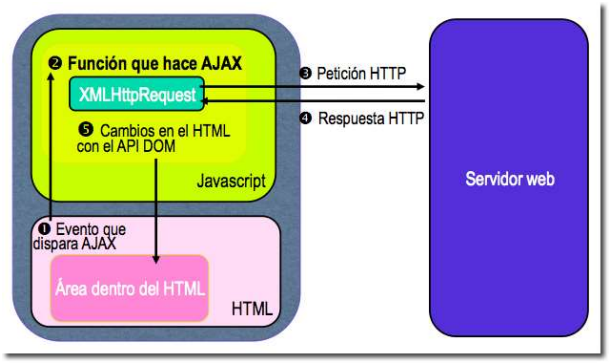

Figura 26. Ciclo de llamadas síncrono en AJAX

AJAX asíncrono

Al realizar una petición síncrona, AJAX bloquea el navegador y se queda a la espera de la respuesta. Para evitar este bloqueo, usaremos el modo **asíncrono**. De este modo, tras realizar la petición, el control vuelve al navegador inmediatamente. El problema que tenemos ahora es averiguar si el recurso solicitado ya esta disponible. Para ello, tenemos la propiedad `readyState`, la cual tenemos que consultar para conocer el estado de la petición. Pero si la consultamos inmediatamente, nos dirá que no ha finalizado. Para evitar tener que crear un bucle infinito de consulta de la propiedad, siguiente el esquema de eventos, usaremos el manejador que ofrece la propiedad `onreadystatechange`.

```
var xhr = new XMLHttpRequest();
xhr.open("GET","fichero.txt", true); ❶
xhr.onreadystatechange = function() {  ❷
  if (xhr.readyState === 4) {  ❸
    alert(xhr.responseText);
  }
};
xhr.send(null);
```

❶ Ahora realizamos la petición de manera asíncrona indicándolo con `true` en el tercer parámetro

❷ Asociamos una función *callback* al evento de cambio de estado

❸ Comprobamos mediante el estado si la petición ha fallado (1), si ha cargado sólo las cabeceras HTTP (2), si está cargándose (3) o si ya esta completa (4).

Además de comprobar el estado de la conexión, debemos comprobar el estado de la respuesta HTTP, para averiguar si realmente hemos obtenido la respuesta que solicitábamos. Para ello, la propiedad `status` nos devolverá el código correspondiente (200 OK, 304 no ha cambiado desde la última petición, 404 no encontrado).

```
xhr.onreadystatechange = function() {
  if (xhr.readyState ===  4) {
    var status = xhr.status;
    if ((status >= 200 && status < 300) || (status === 304)) {
```

```
        alert(xhr.responseText);
      } else {
        alert("Houston, tenemos un problema");
      }
    }
};
```

A modo de esquema, el siguiente gráfico representa las llamadas realizadas mediante una petición asíncrona:

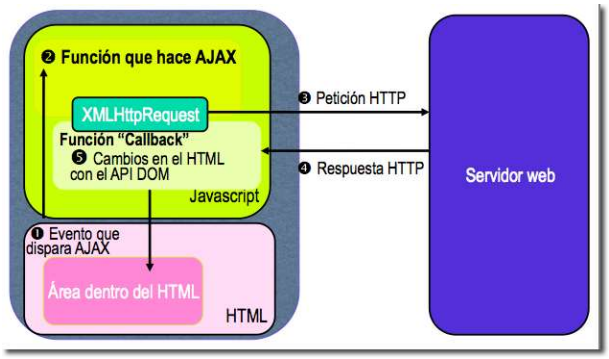

Figura 27. Ciclo de llamadas asíncrono en AJAX

Bucle de Eventos

En la primera sesión comentamos que *JavaScript* es un lenguaje mono-hilo. ¿Cómo se gestionan las peticiones HTTP asíncronas?

Todo el código *JavaScript* corre en un único hilo, mientras que el código que implementa tareas asíncronas no forma parte de *JavaScript* y por tanto es libre de ejecutarse en un hilo separado. Una vez la tarea asíncrona finaliza, el *callback* se sitúa en una cola para devolverle el contenido al hilo principal.

Tras añadir el *callback* a la cola, no hay ninguna garantía de cuanto va a tener que esperar. La cola puede contener sucesos tales como clicks de ratón, teclas pulsadas, respuestas HTTP o cualquier otra tarea asíncrona. El *runtime* de *JavaScript* ejecuta un bucle infinito consistente en llevar el primer elemento de la cola, ejecutar el código que lanza el elemento y volver a comprobar la cola. Este ciclo se conoce como el **bucle de eventos**.

Más información en https://thomashunter.name/blog/the-javascript-event-loop-presentation/

Enviando datos

Cuando vamos a enviar datos, el primer paso es serializarlos. Para ello, debemos considerar qué datos vamos a enviar, ya sean parejas de variable/valor o ficheros, si vamos a emplear el protocolo GET o POST y el formato de los datos a enviar.

HTML5 introduce el objeto `FormData` para serializar los datos y convertir la información a `multipart/form-data`. Se trata de un objeto similar a un mapa, el cual se puede inicializar con un formulario (pasándole al constructor el elemento DOM del formulario) o crearlo en blanco y añadirle valores mediante el método `append()`:

```
var formDataObj = new FormData();

formDataObj.append('uno', 'JavaScript');
formDataObj.append('dos', 'jQuery');
formDataObj.append('tres', 'HTML5');
```

Más información en https://developer.mozilla.org/es/docs/Web/Guide/Usando_Objetos_FormData

Envío mediante GET

A la hora de hacer el envío, mediante el método `send()` de la petición le podemos pasar una cadena compuesta por pares de `variable=valor`, separadas por `&`, con los valores codificados mediante la función `encodeURIComponent`:

```
var valor = "Somos la Ñ";
var datos = "uno=JavaScript&cuatro=" + encodeURIComponent(valor);
// uno=JavaScript&cuatro=Somos%20la%20%C3%91

var xhr = new XMLHttpRequest();
xhr.open("GET", "fichero.txt", true);
// resto de código AJAX
xhr.send(datos);
```

Envío mediante POST

Para poder enviar datos con POST, a la hora de abrir la conexión le indicaremos el método de envío mediante `xhr.setRequestHeader( 'Content-Type', 'application/x-www-form-urlencoded')` o el tipo de contenido deseado (`multipart/form-data`, `text/xml`, `application/json`, ...).

Tras esto, al enviar la petición los datos como una cadena o mediante un objeto `FormData`.

```
var xhr = new XMLHttpRequest();
xhr.open("POST","fichero.txt", true); ❶
xhr.setRequestHeader("Content-Type","application/x-www-form-
urlencoded"); ❷
// Resto de código AJAX
xhr.send("heroe=Batman"); ❸
```

❶ Primer parámetro a `POST`
❷ Indicamos el tipo de contenido que envía el formulario
❸ Le adjuntamos datos de envío como una cadena, o un objeto `FormData`

Eventos

Toda petición AJAX lanza una serie de eventos conforme se realiza y completa la comunicación, ya sea al descargar datos del servidor como al enviarlos. Los eventos que podemos gestionar y el motivo de su lanzamiento son:

- `loadstart`: se lanza al iniciarse la petición
- `progress`: se lanza múltiples veces conforme se transfiere la información
- `load`: al completarse la transferencia
- `error`: se produce un error
- `abort`: el usuario cancela la petición

Todos estos eventos se tienen que añadir antes de abrir la petición.

```
var xhr = new XMLHttpRequest();
xhr.addEventListener('loadstart', onLoadStart, false);
xhr.addEventListener('progress', onProgress, false);
xhr.addEventListener('load', onLoad, false);
xhr.addEventListener('error', onError, false);
xhr.addEventListener('abort', onAbort, false);

xhr.open('GET', 'http://www.omdbapi.com/?s=batman');

function onLoadStart(evt) {
  console.log('Iniciando la petición');
}

function onProgress(evt) {
  var porcentajeActual = (evt.loaded / evt.total) * 100;    ❶
  console.log(porcentajeActual);
}

function onLoad(evt) {
  console.log('Transferencia completada');
}

function onError(evt) {
  console.error('Error durante la transferenciaº');
}

function onAbort(evt) {
  console.error('El usuario ha cancelado la petición');
}
```

❶ Mediante las propiedades `loaded` y `total` del evento, podemos obtener el porcentaje del archivo descargado.

Respuesta HTTP

Si el tipo de datos obtenido de una petición no es una cadena, podemos indicarlo mediante el atributo `responseType`, el cual puede contener los siguientes valores: `text`, `arraybuffer`, `document` (para documentos XML o HTML), `blob` o `json`.

```
var xhr = new XMLHttpRequest();
xhr.open('GET', 'http://expertojava.ua.es/experto/publico/imagenes/logo-
completo.png', true);
```

```
xhr.responseType = 'blob'; ❶

xhr.addEventListener('load', finDescarga, false);
xhr.send();

function finDescarga(evt) {
  if (this.status == 200) {
    var blob = new Blob([this.response], {type: 'img/png'});
    document.getElementById("datos").src = blob;
  }
}
```

❶ Indicamos que el tipo de la imagen es un *blob*

5.2. JSON

Tal como vimos en el módulo de Servicios *REST*, *JSON* es un formato de texto que almacena datos reconocibles como objetos por *JavaScript*. Pese a que *AJAX* nació de la mano de *XML* como el formato idóneo de intercambio de datos, *JSON* se ha convertido en el estándar de facto para el intercambio de datos entre navegador y servidor, ya que se trata de un formato más eficiente y con una sintaxis de acceso a las propiedades más sencilla.

Supongamos que tenemos un archivo de texto (`heroes.json`) con información representada en *JSON*:

```
{
  "nombre": "Batman",
  "email": "batman@heroes.com",
  "gadgets": ["batmovil","batarang"],
  "amigos": [
    { "nombre": "Robin", "email": "robin@heroes.com"},
    { "nombre": "Cat Woman", "email": "catwoman@heroes.com"}
  ]
}
```

Si queremos recuperarla mediante *AJAX*, una vez recuperada la información del servidor, desde *ES5* podemos usar el objeto `JSON` para interactuar con el texto. Este objeto ofrece los siguientes métodos:

- `JSON.stringify(objeto)` obtiene la representación *JSON* de un *objeto* como una cadena, es decir, serializa el objeto, omitiendo todas las funciones, las propiedades con valores `undefined` y propiedades del prototipo.
- `JSON.parse(cadena)` parsea una *cadena JSON* en un objeto *JavaScript*.

```
var batman = { "nombre": "Batman", "email": "batman@heroes.com" };
var batmanTexto = JSON.stringify(batman);
var batmanOBjeto = JSON.parse(batmanTexto);
```

Así pues, volvamos al código AJAX. Una vez recuperada la información del servidor, la transformamos a un objeto mediante `JSON.parse()`.

```
var xhr = new XMLHttpRequest();
xhr.open("GET","heroes.json", true);
xhr.onreadystatechange = function() {
  if (xhr.readyState === 4) {
    var respuesta = JSON.parse(xhr.responseText);  ❶
    alert(respuesta.nombre);  ❷
  }
};
xhr.send(null);
```

❶ Con el texto recibido, lo deserializamos en un objeto respuesta
❷ Si hiciéramos un alert de `resp` tendríamos un objeto, por lo que podemos acceder a las propiedades.

Si quisiéramos enviar datos al servidor en formato JSON, realizaremos una petición POST indicándole como tipo de contenido `application/json`:

```
var xhr = new XMLHttpRequest();
xhr.open("POST","fichero.txt", true);
xhr.setRequestHeader("Content-Type","application/json");
xhr.onreadystatechange = function() {
  // código del manejador
};
xhr.send(JSON.stringify(objeto));
```

eval

Si el navegador no soporta ES5, podemos usar la función `eval(codigo)` que evalúa una cadena como código *JavaScript*. Así pues, podríamos hacer:

```
var textoRespuesta = xhr.responseText;
var objRespuesta = eval( "(" + textoRespuesta + ")" );
```

El problema de `eval` es que además de evaluar *JSON*, también puede evaluar código malicioso, con lo que su uso no se recomienda.

Filtrando campos

Al serializar un objeto mediante `stringify` podemos indicar tanto los campos que queremos incluir como el número de espacios utilizados como sangría del código.

Primero nos centraremos en el filtrado de campos. El segundo parámetro de `stringify` puede ser:

- un *array* con los campos que se incluirán al serializar
- una función que recibe una clave y una propiedad, y que permite modificar el comportamiento de la operación. Si para un campo devuelve `undefined` dicho campo no se serializará.

```
var heroe = {
```

```
  nombre: "Batman",
  email: "batman@heroes.com",
  gadgets: ["batmovil", "batarang"],
  amigos: [
    { nombre: "Robin", email: "robin@heroes.com"},
    { nombre: "Cat Woman", email: "catwoman@heroes.com"}
  ]
};

var nomEmail = JSON.stringify(heroe, ["nombre", "email"]);
var joker = JSON.stringify(heroe, function (clave, valor) {
  switch (clave) {
    case "nombre":
      return "Joker";
    case "email":
      return "joker_" + valor;
    case "gadgets":
      return valor.join(" y ");
    default:
      return valor;
  }
});

console.log(nomEmail); // {"nombre":"Batman","email":"batman@heroes.com"}
console.log(joker); //
 {"nombre":"Joker","email":"joker_batman@heroes.com","gadgets":"batmovil y
 batarang","amigos":[{"nombre":"Joker","email":"joker_robin@heroes.com"},
{"nombre":"Joker","email":"joker_catwoman@heroes.com"}]}
```

En el caso de querer indentar el código, con el tercer parámetro le indicamos con un número (entre 1 y 10) la cantidad de espacios utilizados como sangría, y en el caso de pasarle un carácter, será el elemento utilizado como separador.

```
console.log(JSON.stringify(heroe, ["nombre", "email"]));
console.log(JSON.stringify(heroe, ["nombre", "email"], 4));
console.log(JSON.stringify(heroe, ["nombre", "email"], "<->"));
```

Figura 28. Personalizando la serialización de JSON

❷ Cambiamos la clase CSS de la foto actual para que se posicione detrás y se oculte
❸ La siguiente foto la hacemos transparente, la marcamos como actual
❹ Le añadimos una animación de 1 segundo en la que pasa de transparente a visible
❺ Al terminar la animación, le quitamos la clase de `anterior` para que pase al frente

Deshabilitando los efectos

Si el sistema donde corre nuestra aplicación es poco potente, podemos deshabilitar todos los efectos y animaciones haciendo uso de la propiedad booleana `jQuery.fx.off`:

```
$.fx.off = true;
// Volvemos a activar los efectos
$.fx.off = false;
```

Al estar deshabilitadas, los elementos aparecerán y desaparecer sin ningún tipo de animación.

7.2. AJAX

Todas las peticiones AJAX realizadas con *jQuery* se realizan con el método `$.ajax()` (http://api.jquery.com/category/ajax/).

Para simplificar el trabajo, *jQuery* ofrece varios métodos que realmente son sobrecargas sobre el método `$.ajax()`. Si comprobamos el código fuente de este función podremos ver como es mucho más complejo que lo que estudiamos en la sesión de *JavaScript*.

Los métodos que ofrece *jQuery* son:

Método	Propósito
selector.load(url)	Incrustra el contenido crudo de la *url* sobre el *selector*
$.get(url, callback, tipoDatos)	Realiza una petición GET a la *url*. Una vez recuperada la respuesta, se invocará el *callback* el cual recuperará los datos del *tipoDatos*
$.getJSON(url, callback)	Similar a `$.get()` pero recuperando datos en formato JSON
$.getScript(url, callback)	Carga un archivo *JavaScript* de la *url* mediante un petición GET, y lo ejecuta.
$.post(url, datos, callback)	Realiza una petición POST a la *url*, enviando los datos como parámetros de la petición

Todos estos métodos devuelven un objeto `jqXHR` el cual abstrae el mecanismo de conexión, ya sea un objeto `HTMLHttpRequest`, un objeto `XMLHTTP` o una etiqueta `<script>`.

`$.ajax()`

Ya hemos comentado que el método `$.ajax()` es el que centraliza todas las llamadas AJAX. Pese a que normalmente no lo vamos a emplear directamente, conviene conocer todas las posibilidades que ofrece. Para ello, recibe como parámetro un objeto con las siguientes propiedades:

Propiedad	Propósito
url	URL a la que se realiza la petición

Propiedad	Propósito
type	Tipo de petición (GET o POST)
dataType	Tipo de datos que devuelve la petición, ya sea texto o binario.
success	*Callback* que se invoca cuando la petición ha sido exitosa y que recibe los datos de respuesta como parámetro
error	*Callback* que se invoca cuando la petición ha fallado
complete	*Callback* que se invoca cuando la petición ha finalizado

Por ejemplo, si quisieramos recuperar el archivo `fichero.txt`, realizaríamos una llamada del siguiente modo:

```
$.ajax({
 url: "fichero.txt",
 type: "GET",
 dataType: "text",
 success: todoOK,
 error: fallo,
 complete: function(xhr, estado) {
  console.log("Peticion finalizada " + estado);
 }
});

function todoOK(datos) {
 console.log("Todo ha ido bien " + datos);
}

function fallo(xhr, estado, msjErr) {
 console.log("Algo ha fallado " + msjErr);
}
```

Si quisiéramos incrustar el contenido recibido por la petición, en vez de sacarlo por consola, lo podríamos añadir a una capa o un párrafo:

```
function todoOK(datos) {
 $("#resultado").append(datos);
}
```

`load()`

Si queremos incrustrar contenido proveniente de una URL, con la misma funcionalidad que un *include* estático en JSP, usaremos el método `load(url)`. Por ejemplo:

```
$("body").load("contacto.html");
```

De este modo incluiríamos hasta la cabecera del documento html. Para indicarle que estamos interesados en una parte del documento, podemos indicar que cargue el elemento cuya clase CSS sea `contenido`.

```
$("body").load("contacto.html .contenido");
```

Mediante este método vamos a poder incluir contenido de manera dinámica al lanzarse un evento. Supongamos que tenemos un enlace a `contacto.html`. Vamos a modificarlo para que en vez de redirigir a la página, incruste el contenido:

```
<a href="contacto.html">Contacto</a>
<div id="contenedor"></div>

<script>
$('a').on('click', function(evt) {
 var href = $(this).attr('href'); ❶
 $('#contenedor').load(href + ' .contenido'); ❷
 evt.preventDefault(); ❸
});
</script>
```

❶ Obtenemos el documento que queremos incluir
❷ Incrustamos el contenido del documento dentro de la capa con id `contenedor`
❸ Evitamos que cargue el enlace

Recibiendo información del servidor

Mediante `$.get(url [,datos], callBack(datosRespuesta) [,tipoDatosRespuesta])` se envía una petición GET con `datos` como parámetro. Tras responder el servidor, se ejecutará el *callBack* con los datos recibidos cuyo tipo son del `tipoDatosRespuesta`.

Así pues, el mismo ejemplo visto anteriormente puede quedar reducido al siguiente fragmento:

```
$.get("fichero.txt", function(datos) {
 console.log(datos);
});
```

En el caso de **XML**, seguiremos usando el mismo método pero hemos de tener en cuenta el formato del documento. Supongamos que tenemos el siguiente documento `heroes.xml`:

```
<heroe>
 <nombre>Batman</nombre>
 <email>batman@heroes.com</email>
</heroe>
```

Para poder recuperar el contenido hemos de tener el cuenta que trabajaremos con las funciones DOM que ya conocemos:

```
$.get("heroes.xml", function(datos) {
 var nombre = datos.getElementsByTagName("nombre")[0];
 var email = datos.getElementsByTagName("email")[0];
 var val = nombre.firstChild.nodeValue + " " + email.firstChild.nodeValue;
 $("#resultado").append(val);
},"xml");
```

Si la información a recuperar es de tipo **JSON**, *jQuery* ofrece el método `$.getJSON(url [,datos], callBack(datosRespuesta))`.

Por ejemplo, supongamos que queremos acceder a *Flickr* para obtener las imágenes que tienen cierta etiqueta:

Ejemplo getJSON Flickr - http://jsbin.com/hunape/1/edit?html,js,output

```
var flickrAPI = "http://api.flickr.com/services/feeds/photos_public.gne?
jsoncallback=?";
$.getJSON( flickrAPI, {
 tags: "proyecto víbora ii",
 tagmode: "any",
 format: "json"
}, formateaImagenes);

function formateaImagenes(datos) {
 $.each(datos.items, function(i, elemento) { ❶
  $("<img>").attr("src", elemento.media.m).appendTo("#contenido"); ❷
  if (i === 4) { ❸
   return false;
  }
 });
}
```

❶ la función `$.each(colección, callback(índice, elemento))` recorre el array y realiza una llamada al *callback* para cada uno de los elementos

❷ Por cada imagen, la anexa al id `contenido` construyendo una etiqueta `img`

❸ Limita el número de imágenes a mostrar en 5. Cuando el *callback* de `$.each()` devuelve `false`, detiene la iteración sobre el array

$.each()

La utilidad $.each() se trata de un método auxiliar que ofrece *jQuery* para iterar sobre una colección, ya sea:

- un array mediante `$.each(colección, callback(índice, elemento))`
- un conjunto de selectores con `$(selector).each(callback(índice, elemento))`
- o las propiedades de un objeto mediante `$.each(objeto, callback(clave, valor))`

Se emplea mucho para tratar la respuesta de las peticiones AJAX. Más información en http://api.jquery.com/jquery.each/

Finalmente, en ocasiones necesitamos inyectar código adicional al vuelo. Para ello, podemos recuperar un archivo **JavaScript** y que lo ejecute a continuación mediante el método `$.getScript(urlScript, callback)`. Una vez finalizada la ejecución del script, se invocará al *callback*.

Supongamos que tenemos el siguiente código en `script.js`:

```
console.log("Ejecutado dentro del script");
```

```
$("#resultado").html("<strong>getScript</strong>");
```

Y el código *jQuery* que ejecuta el script:

```
$.getScript("script.js", function(datos, statusTxt) {
 console.log(statusTxt);
})
```

Enviando información al servidor

La función `$.post(url, datos, callback(datosRespuesta))` permite enviar datos mediante una petición POST.

Una singularidad es la manera de adjuntar los datos en la petición, ya sea:

- Creando un objeto cuyos valores obtenemos mediante `val()`.
- Serializando el formulario mediante el método `serialize()`, el cual codifica los elementos del formulario mediante una cadena de texto

Vamos a crear un ejemplo con un formulario sencillo para realizar el envío mediante AJAX:

Ejemplo AJAX POST

```
<form name="formCliente" id="frmClnt" action="#">
<fieldset id="infoPersonal">
 <legend>Datos Personales</legend>
 <p><label for="nombre">Nombre</label>
  <input type="text" name="nombre" id="idNombre" /></p>
 <p><label for="correo">Email</label>
  <input type="email" name="correo" id="idEmail" /></p>
</fieldset>
<p><button type="submit">Guardar</button></p>
</form>
```

Y el código que se comunica con el servidor:

Ejemplo `$.post()`

```
$('form').on('submit', function(evt) { ❶
 evt.preventDefault();  ❷
 // var nom = $(this).find('#inputName').val(); ❸
 var datos = $(this).serialize(); // nombre=asdf&email=asdf
 $.post("/GuardaFormServlet", datos, function (respuestaServidor) { ❹
  console.log("Completado " + respuestaServidor);
 });
});
```

❶ Escuchamos el evento de *submit*
❷ Desactivar el envío por defecto del formulario

❸ Obtener el contenido de los campos, lo cual podemos hacerlo campo por campo como en la línea 3, u obtener una representación de los datos como parámetros de una URL mediante el método `serialize()` como en la línea 4.
❹ Enviar el contenido a un script del servidor y recuperamos la respuesta. Mediante `$.post()` le pasaremos el destino del envío, los datos a envíar y una función *callback* que se llamará cuando el servidor finalice la petición.

Tipos de datos

Ya hemos visto que podemos trabajar con cuatro tipos de datos. A continuación vamos a estudiarlos para averiguar cuando conviene usar uno u otro:

- **Fragmentos HTML** necesitan poco para funcionar, ya que mediante `load()` podemos cargarlos sin necesidad de ejecutar ningún *callback*. Como inconveniente, los datos puede que no tengan ni la estructura ni el formato que necesitemos, con lo que estamos acoplando nuestro contenido con el externo.
- **Archivos JSON**, que permiten estructurar la información para su reutilización. Compactos y fáciles de usar, donde la información es auto-explicativa y se puede manejar mediante objetos mediante `JSON.parse()` y `JSON.stringify()`. Hay que tener cuidado con errores en el contenido de los archivos ya que pueden provocar efectos colaterales.
- **Archivos JavaScript**, ofrecen flexibilidad pero no son realmente un mecanismo de almacenamiento, ya que no podemos usarlos desde sistemas heterogéneos. La posibilidad de cargar scripts *JavaScripts* en caliente permite refactorizar el código en archivos externos, reduciendo el tamaño del código hasta que sea necesario.
- **Archivos XML**, han perdido mercado en favor de JSON, pero se sigue utilizando para permitir que sistemas de terceros sin importar la tecnología de acceso puedan conectarse a nuestros sistemas.

A día de hoy, JSON tiene todas las de ganar, tanto por rendimiento en las comunicaciones como por el tamaño de la información a transmitir.

Manejadores de eventos AJAX

jQuery ofrece un conjunto de métodos globales para interactuar con los eventos que se lanzan al realizar una petición AJAX. Estos métodos no los llamamos dentro de la aplicación, sino que es el navegador el que realiza las llamadas.

Método	Propósito
ajaxComplete()	Registra un manejador que se invocará cuando la petición AJAX se complete
ajaxError()	Registra un manejador que se invocará cuando la petición AJAX se complete con un error
ajaxStart()	Registra un manejador que se invocará cuando la primera petición AJAX comience
ajaxStop()	Registra un manejador que se invocará cuando todas las peticiones AJAX hayan finalizado
ajaxSend()	Adjunta una función que se invocará antes de enviar la petición AJAX
ajaxSuccess()	Adjunta una función que se invocará cuando una petición AJAX finalice correctamente

Recordad que estos métodos son globales y se ejecutan para todas las peticiones AJAX de nuestra aplicación, de ahí que se sólo se adjunten al objeto `document`.

Por ejemplo, si antes de recuperar el archivo de texto registramos todos estos manejadores:

Ejemplo Manejadores AJAX - http://jsbin.com/loqace/edit?js,console

```
$(document).ready(function() {
  $(document).ajaxStart(function () {
    console.log("AJAX comenzando");
  });
  $(document).ajaxStop(function () {
    console.log("AJAX petición finalizada");
  });
  $(document).ajaxSend(function () {
    console.log("Antes de enviar la información...");
  });
  $(document).ajaxComplete(function () {
    console.log("Todo ha finalizado!");
  });
  $(document).ajaxError(function (evt, jqXHR, settings, err) {
    console.error("Houston, tenemos un problema: " + evt + " - jq:" +
 jqXHR + " - settings :" + settings + " err:" + err);
  });
  $(document).ajaxSuccess(function () {
    console.log("Parece que ha funcionado todo!");
  });

  getDatos();
});

function getDatos() {
 $.getJSON("http://www.omdbapi.com/?s=batman&callback=?", todoOk);
}

function todoOk(datos) {
 console.log("Datos recibidos y adjuntándolos a resultado");
 $("#resultado").append(JSON.stringify(datos));
}
```

Tras ejecutar el código, por la consola aparecerán los siguientes mensajes en el orden en el que se ejecutan:

Figura 46. Eventos globales AJAX

```
      rechazar(new Error("No se ha podido recuperar los datos de " +
 url));
    }, tiempo);
  });

  return Promse.race([datosServer, datosCache, fallo]);
}
```

Se trata de una solución simplificada, donde se ha omitido el código de `buscarEnCache`, y no se han planteado todos los escenarios posibles.

Autoevaluación

¿Qué pasaría se la petición a los datos del servidor falla inmediatamente debido a un fallo de red? [17]

Librerías de programación reactiva como RxJS, Bacon.js y Kefir.js están diseñadas especificamente para estos escenarios

8.3. Fetch API

Aprovechando las promesas, ES6 ofrece el Fetch API para realizar peticiones AJAX que directamente devuelvan un promesa. Es decir, ya no se emplea el objeto `XMLHttpRequest`, el cual no se creó con AJAX en mente, sino una serie de métodos y objetos diseñados para tal fin. Al emplear promesas, el código es más sencillo y limpio, evitando los *callbacks* anidados.

Actualmente, el API lo soportan tanto *Google Chrome* como *Mozilla Firefox*. Podéis comprobar el resto de navegadores en http://caniuse.com/#search=fetch

El objeto `window` ofrece el método `fetch`, con un primer argumento con la URL de la petición, y un segundo opcional con un objeto literal que permite configurar la petición:

Ejemplo Fetch API

```
// url (obligatorio), opciones (opcional)
fetch('/ruta/url', {
 method: 'get'
}).then(function(respuesta) {

}).catch(function(err) {
 // Error :(
});
```

Hola *Fetch*

A modo de ejemplo, vamos a reescribir el ejemplo de la sesión 5 que hacíamos uso de AJAX, pero ahora con la *Fetch API*. De esta manera, el código queda mucho más concreto:

```
fetch('http://www.omdbapi.com/?s=batman', {
    method: 'get'
```

[17] La promesa `datosServer` se rechazaría y por consiguiente no se comprobaría la caché

```
}).then(function(respuesta) {
  if (!respuesta.ok) {
    throw Error(respuesta.statusText);
  }
  return respuesta.json();
}).then(function(datos) {
    var pelis = datos.Search;
    for (var numPeli in pelis) {
        console.log(pelis[numPeli].Title + ": " + pelis[numPeli].Year);
    }
}).catch(function(err) {
 console.error("Error en Fetch de películas de Batman", err);
});
```

Como podemos observar, haciendo uso de las promesas, podemos encadenarlas y en el último paso comprobar los errores.

Por ejemplo, podemos refactorizar el control de estado y extraelo a una función:

```
function estado(respuesta) {
  if (respuesta.ok) {
    return Promise.resolve(respuesta);
  } else {
    return Promise.reject(new Error(respuesta.statusText));
  }
}
```

De este modo, nuestro código quedaría así:

```
fetch('http://www.omdbapi.com/?s=batman', {
  method: 'get'
}).then(estado)
.then(function (respuesta) {
  return respuesta.json();
}).then(function(datos) {
  var pelis = datos.Search;
  for (var numPeli in pelis) {
    console.log(pelis[numPeli].Title + ": " + pelis[numPeli].Year);
  }
}).catch(function(err) {
  console.error("Error en Fetch de películas de Batman", err);
});
```

Cabeceras

Una ventaja de este API es la posibilidad de asignar las cabeceras de las peticiones mediante el objeto `Headers()`, el cual tiene una estructura similar a una mapa. A continuación se muestra un ejemplo de como acceder y manipular las cabeceras mediante los métodos `append`, `has`, `get`, `set` y `delete`:

```
var headers = new Headers();
```

```
headers.append('Content-Type', 'text/plain');
headers.append('Mi-Cabecera-Personalizada', 'cualquierValor');

headers.has('Content-Type'); // true
headers.get('Content-Type'); // "text/plain"
headers.set('Content-Type', 'application/json');

headers.delete('Mi-Cabecera-Personalizada');

// Add initial values
var headers = new Headers({ ❶
  'Content-Type': 'text/plain',
  'Mi-Cabecera-Personalizada': 'cualquierValor'
});
```

Para usar las cabeceras, las pasaremos como parámetro de creación de una petición mediante una instancia de `Request`:

```
var peticion = new Request('/url-peticion', {
  headers: new Headers({
    'Content-Type': 'text/plain'
  })
});

fetch(peticion).then(function() { /* manejar la respuesta */ });
```

Petición

Para poder configurar toda la información que representa una petición se emplea el objeto `Request`. Este objeto puede contener las siguientes propiedades:

- `method`: `GET`, `POST`, `PUT`, `DELETE`, `HEAD`
- `url`: URL de la petición
- `headers`: objeto Headers con las cabeceras asociadas
- `body`: datos a enviar con la petición
- `referrer`: *referrer* de la petición
- `mode`: `cors`, `no-cors`, `same-origin`
- `credentials`: indica si se envian *cookies* con la petición: `include`, `omit`, `same-origin`
- `redirect`: `follow`, `error`, `manual`
- `integrity`: valor integridad del subrecurso
- `cache`: tipo de cache (`default`, `reload`, `no-cache`)

Con estos parámetros, una petición puede quedar así:

```
var request = new Request('/heroes.json', {
  method: 'get',
  mode: 'cors',
```

```
  headers: new Headers({
    'Content-Type': 'text/plain'
  })
});

fetch(request).then(function() { /* manejar la respuesta */ });
```

Realmente, el método `fetch` recibe una URL a la que se envía una petición, y un objeto literal con la configuración de la petición, por lo que el código queda mejor así:

```
fetch('/heroes.json', {
  method: 'GET',
  mode: 'cors',
  headers: new Headers({
    'Content-Type': 'text/plain'
  })
}).then(function() { /* manejar la respuesta */ });
```

Enviando datos

Ya sabemos que un caso muy común es enviar la información de un formulario vía AJAX.

Para ello, además de configurar que la petición sea `POST`, le asociaremos en la propiedad `body` un `FormData` creado a partir del identificador del elemento del formulario:

```
fetch('/submit', {
  method: 'post',
  body: new FormData(document.getElementById('formulario-cliente'))
});
```

Si lo que queremos es enviar JSON al servidor, en la misma propiedad, le asociamos un nuevo objeto JSON:

```
fetch('/submit-json', {
  method: 'post',
  body: JSON.stringify({
    email: document.getElementById('email').value
    comentarios: document.getElementById('comentarios').value
  })
});
```

O si queremos enviar información en formato URL:

```
fetch('/submit-urlencoded', {
  method: 'post',
  headers: {
    "Content-type": "application/x-www-form-urlencoded; charset=UTF-8"
  },
  body: 'heroe=Batman&nombre=Bruce+Wayne'
});
```

Respuesta

Una vez realizada la petición, dentro del manejador, recibiremos un objeto `Response`, compuesto de una serie de propiedades que representan la respuesta del servidor, de las cuales extraremos la información:

- `type`: indican el origen de la petición. Dependiendo del tipo, podremos consultar diferente información:
  - `basic`: proviene del mismo origen, sin restricciones.
  - `cors`: acceso permitido a origen externo. Las cabeceras que se pueden consultar son `Cache-Control`, `Content-Language`, `Content-Type`, `Expires`, `Last-Modified`, y `Pragma`
  - `opaque`: origen externo que no devuelve cabeceras CORS, con lo que no podemos leer los datos ni visualizar el estado de la petición.
- `status`: código de estado (200, 404, etc.)
- `ok`: Booleano que indica si la respuesta fue exitosa (en el rango 200-299 de estado)
- `statusText`: código de estado (`OK`)
- `headers`: objeto Headers asociado a la respuesta.

Además, el objeto `Response` ofrece los siguientes métodos para crear nuevas respuestas con diferente código de estado o a un destino distino:

- `clone()`: clona el objeto *Response*
- `error()`: devuelve un nuevo objeto *Response* asociado con un error de red.
- `redirect()`: crea una nueva respuesta con una URL diferente

Finalmente, para transformar la respuesta a una promesa con un tipo de dato del cual extraer la información de la petición, tenemos los siguientes métodos:

- `arrayBuffer()`: Devuelve una promesa que se resuelve con un *ArrayBuffer*.
- `blob()`: Devuelve una promesa que se resuelve con un *Blob*.
- `formData()`: Devuelve una promesa que se resuelve con un objeto *FormData*.
- `json()`: Devuelve una promesa que se resuelve con un objeto *JSON*.
- `text()`: Devuelve una promesa que se resuelve con un texto (*USVString*).

Veamos estos métodos en funcionamiento.

Parseando la respuesta

A día de hoy, el estandar es emplear el formato JSON para las respuestas. En vez de utilizar `JSON.parse(cadena)` para transformar la cadena de respuesta en un objeto, podemos utilizar el método `json()`:

```
fetch('heroes.json').then(function(response) {
  return response.json();
}).then(function(datos) {
  console.log(datos); // datos es un objeto JavaScript
```