JAVASCRIPT
AJAX PASO A PASO

—

José Luis Rojo Sánchez

INTRODUCCIÓN

Aprende a crear aplicaciones web mucho más rápidas e interactivas usando la técnica de Ajax.

Tecnologías

Para entender toda su magnitud deberás usar HTML, CSS, XML, XSLT, JAVASCRIPT, XHR. entornos de programación de servidor estilo PHP y formatos de intercambio de datos como JSON.

Índice de Contenidos

1 Ajax

La comunicación de scripts con el servidor, ya era posible desde 1998. A esta técnica se la conocía como **Remote Scripting** y se realizaba usando JavaScript a través de intermediarios como eran los Applets de Java o películas de Flash.

Pero no fué hasta 2005 cuando **Jesse James Garrett**[1] presentó **AJAX** por primera vez y donde explicaba cómo funcionaba esta nueva técnica haciendo uso de otras tecnologías ya existentes (xHTML+CSS, DOM, XML, XSLT y JSON, XHR y JavaScript).

Ajax (Asynchronous JavaScript and XML) es por tanto una técnica que consiste en realizar peticiones de datos sin necesidad de recargar la página, reduciendo drásticamente la carga y resultando una mejor experiencia para el usuario.

La clave para que esta técnica funcionara fue el objeto **XMLHttpRequest** (**XHR**), que fué inventado por Microsoft y posteriormente implementado por otros navegadores.

Inicialmente este estilo de comunicaciones tenía que realizarse mediante una serie de Hacks ocultos, en su mayoría utilizando marcos o iframes.

XHR introdujo una interfaz para poder realizar estas peticiones y así poder evaluar las respuestas. Esto permitió realizar peticiones asíncronas para solicitar o enviar información al servidor, no teniendo que refrescar la página para poder recibir los datos.

XHR se encargaba de recibir dichos datos y estos eran insertados en la página a través del DOM (Document Object Model).

[1] https://adaptivepath.org/ideas/ajax-new-approach-web-applications/

1.1 XMLHttpRequest - Level 1

Antes de adentrarnos en el uso actual del objeto XHR (XMLHttpRequest) deberíamos conocer su evolución con el paso del tiempo.

Microsoft Internet Explorer 5.0 fué el primer navegador en introducir el objeto XHR y lo hizo posible usando un objeto ActiveX incluido como parte de la librería MSXML.

Este permitía el uso de tres versiones del objeto XHR y estas podían usarse en el navegador: MSXML2.XMLHttp, MSXML2.XMLHttp.3.0 y MSXML.Http.6.0.

Por lo que esta fué la primera versión del objeto XHR usada hasta la llegada de Internet Explorer 8. Veamos cómo se inicializaba el objeto en sus inicios.

```
function createXHR() {
   if (typeof arguments.callee.activeXString != "string") {
      var versions = ["MSXML2.XMLHttp.6.0", "MSXML2.XMLHttp.3.0",
"MSXML2.XMLHttp"], i, len;
      for (i=0, len=versions.length; i < len; i++){
         try {
            new ActiveXObject(versions[i]);
            arguments.callee.activeXString = versions[i];
            break;
         } catch (ex){
            // skip
         }
      }
   }
   return new ActiveXObject(arguments.callee.activeXString);
}
```

Ya en 2008 las nuevas versiones de Internet Explorer 8 y otros navegadores de la época como eran Firefox, Opera, Chrome en su versión Beta y Safari daban soporte al objeto XHR y este podría ser inicializado usando el constructor XMLHttpRequest.

```
var xhr = new XMLHttpRequest();
```

Por lo que si necesitabas dar soporte al objeto XHR en versiones anteriores a la 8 necesitabas incluir unos condicionantes para poder comprobar el tipo de navegador usado por el usuario. Fueron momentos interesantes para todo programador.

```
function createXHR(){
    if (typeof XMLHttpRequest != "undefined") {
        return new XMLHttpRequest();
    } else if (typeof ActiveXObject != "undefined") {
        // código para versiones antiguas IE <=7
    } else {
        throw new Error("No XHR object available.");
    }
}
```

Actualmente ya no necesitamos este tipo de códigos, es raro encontrarse navegadores con más de 10 años de antigüedad pero aún así podemos realizar una comprobación básica.

```
function createXHR() {
    try {
        return new XMLHttpRequest();
    } catch(e) {
        // document.getElementById("error").innerHTML = e.message;
        return "Necesita instalar un navegador más moderno.";
    }
}
```

1.1.1 Uso del objeto XHR

Como hemos comentando anteriormente, la estandarización del objeto XHR nos permite usarlo actualmente en cualquier navegador. Por lo que no debería ser necesario crear una función para comprobar si está presente.

```
var xhr = new XMLHttpRequest();
```

Una vez inicializado el objeto XHR realizamos una llamada al método open(), que aceptará 3 argumentos:

- El tipo de petición que se está realizando (GET, HEAD ó POST).
- La Url de la petición.
- Y un valor booleano (true o false), que si es true indicará que la petición debe ser asíncrona.

```
xhr.open("get", "test.php", false);
```

En este caso estamos realizando una nueva petición GET, síncrona, al fichero test.php.

También podemos hacer uso del envío de parámetros a través de la URL.

```
xhr.open("get", "test.php?param=1&param=2&param=3", false);
```

Debemos tener en cuenta que la dirección del fichero solicitado es relativa al fichero que solicita la petición y lo segundo y no menos importante es que el método open() no será el que realiza la petición sino que sólo la prepara para poder enviarla.

Uno de los errores más comunes es codificar erróneamente estos parámetros. Cada parámetro y su valor correspondiente son codificados usando el método encodeURIComponent().

Para poder iniciar la petición debemos hacer uso del método send()

```
xhr.send(null);
```

El método send() acepta un argumento que serán los datos enviados al documento solicitado. En el caso que no necesitemos pasar parámetros deberíamos indicar null ya que este argumento es obligatorio en algunos navegadores.

Cuando la solicitud es “síncrona”, el código JavaScript esperará hasta que el servidor de una respuesta y así poder continuar con la carga del documento.

Las peticiones “asíncronas” son más importantes y más populares ya que permiten al código JavaScript seguir ejecutándose sin necesidad de esperar a que este termine.

1.1.2 Propiedades

Cuando la respuesta es completada, las propiedades del objeto XHR son rellenadas con los datos recibidos.

Veamos la función de cada una de ellas:

- onreadystatechange - Un manejador de eventos que es llamado cuando la propiedad readyState cambia.
- readyState - El estado actual de la petición (0 a 4)
- response - Retorna el tipo de datos que retorna la respuesta donde los más importantes son (arraybuffer ó binary si son datos binarios, document si es un documento HTML o xML, json, text)
- responseText - Igual que responseText salvo que los datos que retorna son en formato texto.
- responseXML - Contiene el XML DOM con los datos de la respuesta dependiendo si estos son text/html o application/xml.
- responseURL - La URL absoluta del fichero solicitado.
- status - El estado HTTP de la respuesta (200, 404, etc ..).
- statusText - La descripción del estado de la respuesta (OK o Not Found).
- timeout - Cantidad de milisegundos que se esperará hasta que la respuesta sea terminada.
- ontimeout - El manejador de eventos de timeout.
- upload - Indica que existe un proceso de carga.
- withCredentials - En un valor booleano que indicará si las solicitudes de acceso deben ser realizadas mediante cookies, cabeceras personalizadas o certificados TLS.

Como es impredecible saber cuándo terminará el script y cuándo se notificará la respuesta de su finalización podemos hacer uso del evento onreadystatechange que comprueba las propiedades de readystate par determinar el estado de la respuesta.

Valor	valores de **readystate**
0	Inicializado pero aún sin ser llamada.
1	El método open() es llamado pero no ha sido enviado.
2	El método send() es llamado pero la respuesta aún no se ha recibido.

3	Recibiendo. Las cabeceras HTTP han sido recibidas pero los datos aún no.
4	Cargado. Recepción de los datos y respuesta completada.

En una petición asíncrona, para un programador el único estado que nos interesa comprobar es el **4**.

Veamos un ejemplo de cómo trabajar con el evento onreadystatechange para así poder comprobar el estado de la petición.

En el ejemplo realizamos una solicitud a un fichero y este nos devolverá el contenido para así poder imprimirlo en pantalla.

```
/*
* @function
* @desc muestra el contenido de un fichero
* @return text/plain
*/
function mostrarContenido() {
  // XHR Object (AJAX)
  var xhr = new XMLHttpRequest();
  var messageElement = document.getElementById("message");

  xhr.onreadystatechange = function() {
     if ( xhr.readyState == 4 ) { // readyState = 4
        if ( xhr.status == 200 ) { // http = 200
           console.log(this.getAllResponseHeaders());
           console.log(this.responseText);
           messageElement.innerHTML = this.responseText;
        } else {
           console.log("Error:" + this.status);
           messageElement.innerHTML = "Se ha producido un error!";
        }
     }
  };

  var uploadMethod = "get";
  var uploadUrl = "php/contenido.php";
  var uploadType = true;

  xhr.open(uploadMethod, uploadUrl, uploadType);
```

```
  xhr.send(null);
}
```

El documento HTML contendrá un botón con un evento onclick que llamará a la función mostrarContenido() y el resultado será mostrado en la capa message.

```
<button onclick="mostrarContenido()">Mostrar contenido fichero</button>
<div id="message"></div>
```

1.1.3 Métodos

Hasta el momento hemos usado los métodos open() y send() que preparan e inician la petición, pero existen otros métodos que iremos viendo más adelante.

- open() - Se encarga de preparar la petición.
- send() - Inicia la petición.
- abort() - Nos permitirá cancelar una petición asíncrona antes de que una respuesta sea recibida.
- getAllResponseHeaders() - Recupera todas las cabeceras HTTP recibidas por el servidor.
- getResponseHeader() - Acepta un parámetro que será la cabecera que queremos recuperar.
- setRequestHeader() - Acepta dos parámetros y podemos indicar una nueva cabecera y su valor.
- overrideMimeType() que los veremos en el tema XMLHttpRequest Level 2.

1.1.4 Cabeceras HTTP

Con cada solicitud HTTP se generan unas cabeceras con meta datos, las cuales serán utilizadas por los métodos y propiedades del objeto XHR.

Cuando se inicia una petición XHR se envían las siguientes cabeceras:

- Accept - El tipo de contenido que el navegador puede manejar.
- Accept-Charset - la codificación que maneja el navegador.

- Accept-Encoding - Compresión de la codificación.
- Accept-Language - El idioma del navegador.
- Connection - El tipo de conexión que se realiza con el servidor.
- Cookie - Cookies de la página.
- Host - El dominio de la página que realiza la petición.
- Referer - URL de la página que realiza la petición.
- User-Agent - El tipo de navegador.

Para poder incluir cabeceras personalizadas podemos hacer uso de método setRequestHeader(), el cual acepta 2 argumentos (el nombre de la cabecera y su valor).

```
xhr.open("get", "example.txt", true);
xhr.setRequestHeader("miCabecera", "miValor");
xhr.send(null);
```

Para consultar una cabecera haremos uso del método getResponseHeader(), indicando como argumento el nombre de la cabecera.

```
var miCabecera = xhr.getResponseHeader("content-typea");
```

Para consultar todas las cabeceras haremos uso del método getAllResponseHeader() y retornará algo muy parecido a esto.

```
console.log(xhr.getAllResponseHeaders());
```

```
date: Thu, 14 Mar 2019 19:29:02 GMT
server: nginx
connection: keep-alive
x-powered-by: PHP/7.0.33, PleskLin
transfer-encoding: chunked
content-type: application/json
```

En este caso estamos recibiendo los datos en formato json.

1.2 XMLHttpRequest - Level 2

La popularidad del objeto XHR permitió la creación de una especificación oficial de la W3C, donde **XMLHttpRequest Level 1** definía la implementación actual del objeto XHR y **XMLHttpRequest Level 2**[2] permitía una evolución ya mejorada del objeto XHR.

1.2.1 El tipo FormData

La serialización de datos de un formulario tiene una importancia relevante en cualquier aplicación moderna y es lo que introduce la especificación XMLHttpRequest Level 2

Se introduce el objeto FormData() que nos hace la vida más fácil serializando formularios y así poder enviarlos a través del objeto XHR.

```
var data = new FormData();
data.append("email", "jose@artegrafico.net");
```

Una vez creada la instancia del objeto FormData(), podemos hacer uso del método append() el cual acepta 2 argumentos (una clave y un valor). Podremos incluir tantos como deseemos.

Vamos a crear un ejemplo de petición usando los campos de un formulario de acceso y pasándole los datos al constructor FormData().

En primer lugar crearemos el formulario HTML con los campos típicos de un formulario de acceso (email y clave) y una capa llamada message donde mostraremos los datos devueltos por el servidor sino.

```
<aside class="col-3">
   <form name="contacto" id="contacto" method="post" action=""
onsubmit="enviar(this); return false;">
      <div class="form-group">
         <label for="nombre">Nombre: </label>
         <input type="text" name="email" id="email" class="form-control" />
```

[2] https://xhr.spec.whatwg.org/

```
      </div>
      <div class="form-group">
         <label for="passwd">Password: </label>
         <input type="text" name="passwd" id="passwd"  class="form-control" />
      </div>
      <p>
         <input type="submit" value="Login" class="btn btn-primary" />
      </p>
      <div id="message"></div>
   </form>
</aside>
```

El siguiente paso es crear la función ECMAScript que se encargará de enviar los datos del formulario al servidor para que estos sean comprobados.

```
/*
 * @function enviar()
 * @desc send form data to server
 * @return array json
 */
function enviar() {

   var messageElement = document.getElementById('message');

   // XHR Object (AJAX)
   var xhr = new XMLHttpRequest();

   xhr.onreadystatechange = function(res){
      if (xhr.readyState == 4){
         if (xhr.status == 200){

            // parse json response {status, message}
            var res = JSON.parse(this.response);

            console.log(this.getAllResponseHeaders());
            console.log(this.responseText);
            console.log("email: "+res.email);
            console.log("password: "+res.passwd);
            messageElement.innerHTML = this.response;
         } else {
```

```
            alert('Error: '  + xhr.status);
        }
    }
};

// prepare request
xhr.open('post', 'php/login.php', true);

// prepare FormData()
var form = document.getElementById('contacto');
data = new FormData(form);
data.append('id', "1234"); // campo adiccional

// Send Request
xhr.send(data);

}
```

Por último el fichero login.php recibe los datos y los comprobará en la base de datos. En el ejemplo solo retornamos los datos en formato json y así para entender su funcionamiento.

```
<?php
header('Content-Type: application/json');
// check data vs DB

// return data to debug
echo json_encode([
    'email' => filter_input(INPUT_POST, 'email', FILTER_SANITIZE_EMAIL),
    'passwd' => filter_input(INPUT_POST, 'passwd'),
    'id' => filter_input(INPUT_POST, 'id', FILTER_SANITIZE_NUMBER_INT)
]);
```

Una de las comodidades de FormData() es que no necesitamos especificar las cabeceras del objeto XHR. El Objeto XHR reconocerá todos los tipos de datos pasados y se encargará de configurarlas correctamente.

FormData() actualmente es soportado por todos los navegadores.

1.2.2 Timeouts

La propiedad timeout fué introducida en Microsoft Internet Explorer 8 y permitía indicar el número de milisegundos que la petición debería esperar antes de recibir un error y abortar la solicitud.

Por lo que sin establecemos la propiedad timeout y la respuesta no es recibida en los milisegundos indicados, el evento timeout es ejecutado y el manejador del evento ontimeout será llamado.

Esta funcionalidad fué incluida posteriormente en la especificación XMLHttpRequest Level 2.

```
xhr.open('get', 'descarga.php', true);
xhr.timeout = 5000;  // 5 segundos
xhr.ontimeout = function(){
   alert('Se ha producido un error procesando la petición');
};
xhr.send(null);
```

1.2.3 El método overrideMimeType()

Firefox introdujo el método overrideMimeType() con el propósito de poder anular los tipos MIME de una respuesta XHR. Más tarde fué incluida a XMLHttpRequest Level 2.

Si el servidor retorna el tipo MIME text/html y este contiene datos en XML, los datos no serán tratados correctamente. Como solución utilizaremos el método overrideMimeType()

```
var xhr = new XMLHttpRequest();
xhr.open('get','mifichero.php',true);
xhr.overrideMimeType('text/xml');
xhr.send(null);
```

De esta manera forzamos al objeto XHR a tratar los datos devueltos como XML.

1.3 Eventos de progreso

Los *eventos de progreso*[3] son una especificación de la W3C en su versión 1.0 de abril del 2014. En ella se definen una serie de eventos que permiten gestionar la comunicación cliente-servidor.

Hay 7 tipos de eventos de progreso que podremos gestionar a través de la propiedad upload.

La propiedad upload retornará un objeto llamado XMLHttpRequestUpload que permite ser observado para así poder monitorizar el proceso de carga.

Un procedimiento o manejador de eventos (event listener) se encargará de escuchar cuando se produce el evento.

Evento	Evento de Escucha	Descripción
loadstart	onloadstart	Se dispara cuando el primer byte de la respuesta ha sido recibido.
progress	onprogress	Se dispara repetidamente mientras la respuesta está siendo recibida.
error	onerror	Se dispara cuando ocurre un error durante la recepción de la respuesta.
abort	onabort	Se dispara cuando la conexión ha sido terminada.
timeout	ontimeout	La carga se ha terminado porque el tiempo ha expirado.
load	onload	Se dispara cuando la respuesta ya ha sido recibida.

[3] https://www.w3.org/TR/progress-events/

loadend	onloadend	Se dispara cuando la comunicación ya ha sido completada y después de ser dispararse un evento de error, abort o load.

Cada petición siempre comenzará con un evento loadstart, seguido de uno o más eventos progress, a continuación podrían ocurrir algún tipo de evento error, abort, timeout o load, para terminar con un único evento loadend que indica que el progreso ha terminado

1.3.1 El evento onload

El evento load se dispara tan pronto como la respuesta ha sido recibida, evitando la necesidad de tener que chequear la propiedad readyState que vimos en ejemplos anteriores.

onload recibe un objeto cuyo objetivo es establecer una instancia del objeto XHR donde estarán todas sus propiedades y métodos.

Veamos el ejemplo de login usando el evento onload

```
/*
 * @function enviar()
 * @desc send form data to server
 * @return array json
 */
function enviar() {

   var messageElement = document.getElementById('message');

   // XHR Object (AJAX)
   var xhr = new XMLHttpRequest();

   xhr.onload = function (res) {
      if (xhr.status == 200) {
         // parse json response {status, message}
         var res = JSON.parse(this.response);

         console.log(this.getAllResponseHeaders());
```

```
            console.log(this.response);
            console.log("email: " + res.email);
            console.log("password: " + res.passwd);
            messageElement.innerHTML = this.response;
        } else {
            alert('Error: ' + xhr.status);
        }
    };

    // prepare request
    xhr.open('post', 'php/login.php', true);

    // prepare FormData()
    var form = document.getElementById('contacto');
    data = new FormData(form);
    data.append('id', "1234"); // campo adiccional

    // Send Request
    xhr.send(data);
}
```

Cuando la respuesta desde el servidor ha sido recibida, independientemente del estado de esta, el evento load se dispara, por lo que necesitamos chequear la propiedad status para determinar si los datos están disponibles.

1.3.2 El evento progress

Otra mejora que incluyó Mozilla al objeto XHR es el evento progress, que se dispara periódicamente mientras el navegador recibe o envía los datos.

El manejador de evento onprogress escucha el evento progress, que contiene las siguientes propiedades:

- loaded - El número de bytes transferidos desde que se ha iniciado la operación.
- total - El número total de bytes que se recibirán cuyo valor está almacenado en la cabecera Content-Length.
- lengthComputable - es un valor booleano que indica si se conoce el tamaño total de la transferencia de datos a realizar.
- position - contiene el número de bytes que han sido recibidos
- cancelable - Es un valor booleano que indica si es posible cancelar el evento.
- type - Tipo de evento (será progress).
- target - El nivel más alto del evento en el DOM.

```
// XHR Object (AJAX)
var xhr = new XMLHttpRequest();

xhr.onload = function(res) {

   // parse json response {status, message}
   var messageElement = document.getElementById('message');
   var res = JSON.parse(this.response);

   if (xhr.status == 200) {
      // hacer algo
      console.log(this.responseText);
   } else {
      console.log('error: '+xhr.status);
   }
}
xhr.onprogress = function(event) {
   if (event.lengthComputable) {
      messageElement.innerHTML = "Recibidos "+ event.loaded + " de " + event.total +
"bytes";
```

```
    }
}
xhr.open('get', 'upload.php', true);
xhr.send(null);
```

Para una correcta ejecución, el manejador del evento onprogress debe ser incluido antes del método open()

1.3.3 El evento onerror

Cuando necesitemos tratar errores a través del evento onerror hay que tener en cuenta que los errores HTTP como puede ser que no se encuentre el fichero solicitado, no son considerados como tal por lo que las respuestas del servidor a través del evento onload siempre serán ejecutadas.

Como alternativa deberíamos usar onreadystatechange.

Para entender los eventos de progreso vamos a crear un cargador de ficheros más complejo para así poder entender su funcionamiento.

Lo primero que vamos a hacer es crear el documento HTML

```
<!DOCTYPE html>
<html>
<head>
  <link rel="stylesheet" type="text/css" href="css/upload.css" />
  <link rel="stylesheet" type="text/css"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min.css" />
  <script type="text/javascript" src="js/upload.js"></script>
</head>

<body>
  <div class="container">
    <aside class="aside col-6">
      <form id="upform">
        <div class="form-group">
          <label for="upfile">Selecciona un fichero ...</label>
          <input type="file" class="upfile form-control-file" id="upfile" required />
          <div id="bar">
```

```
                <div id="percent">0%</div>
              </div>
            </div>
            <div class="form-group">
              <div id="message" class="alert"></div>
              <input type="button" value="Upload" onclick="save();" class="btn
btn-primary" />
            </div>
          </form>
        </aside>
    </div>
</body>
</html>
```

Ahora crearemos nuestro script con la función save() y el resto de componentes:

```
/* @function save()
 * @desc on press upload buttom send data to server
 * @return json array
 */
function save() {

    // debug mode
    var debug = true;
    var debugProgress = false;
    // bar div
    var barElement = document.getElementById("bar");
    // bar percent div
    var percentElement = document.getElementById("percent");
    // messages div
    var messageElement = document.getElementById('message');
    // input file
    var upfile = document.getElementById("upfile").files;
    // hide bar on default
    barElement.style.display = "none";

    // status bar properties
    var bar = {
        percent: 0, // default 0
        draw: function () { // function to change attr bar
```

```
            var maxwidth = 100,
            width = Math.ceil(this.percent * maxwidth);
            barElement.style.width = width + "%"; // increment with
            percentElement.innerHTML = width + "% cargado"; // show percent loaded
        }
    };

    // check if file input is selected
    if (upfile.length == 0) {

        // add class error and show message
        messageElement.classList.add("alert-danger");
        messageElement.style.display = "block";
        messageElement.innerHTML = "* Seleccione un fichero a cargar.";

    } else {

        // XHR Object (AJAX)
        var xhr = new XMLHttpRequest();
        // FormData Object to serialized data
        var data = new FormData();
        // append method to add file
        data.append("upfile", upfile[0]);

        // open() method configuration
        var uploadMethod = "POST";
        var uploadUrl = "ajax-upload.php";
        var uploadType = true;

        // response data received from file
        // onload is executed, only if response file exist
        xhr.onload = function (res) {

            if (this.status != 200) {
                messageElement.classList.add("alert-danger");
                messageElement.style.display = "block";
                messageElement.innerHTML = "Se ha producido un error (consulte con el
administrador).";
            } else {

                // parse json response {status, message}
```

```
        var res = JSON.parse(this.response);

        // on upload complete, show server message from json data
        // if response server status = true
        if (res.status) { // boolean type
            messageElement.classList.remove("alert-danger");
            messageElement.classList.add("alert-success");
            messageElement.style.display = "block";
            messageElement.innerHTML = res.message;
        } else {
            messageElement.classList.remove("alert-success");
            messageElement.classList.add("alert-danger");
            messageElement.style.display = "block";
            messageElement.innerHTML = res.message;
        }

        // if debug mode true
        if (debug) {

            var debugResponse = {
                "responseURL": this.responseURL,
                "responseType": this.responseType,
                "responseText": this.responseText,
                "response": this.response,
                "status": this.status,
                "statusText": this.statusText,
                // parsed json data
                "json message": res.message,
                "json status": res.status,
                "json origin": res.origin,
                "json debug": res.debug
            };
            console.log(this.getAllResponseHeaders());
            console.log(JSON.stringify(debugResponse, null, 4));
        }
    }
};

xhr.onerror = function (e) {
    console.error("error:".xhr.statusText);
};
```

```
    // progress bar configuration
    // onprogress send multiple states
    if (xhr.upload) {

        xhr.upload.onloadstart = function (event) {
            barElement.style.display = "block"; // show progress bar
            bar.percent = 0;
            bar.draw();
        };
        xhr.upload.onprogress = function (event) {
            if (event.lengthComputable) {
                bar.percent = event.loaded / event.total;
                bar.draw();

                if (debugProgress) {
                    var debugResponse = {
                        "loaded": event.loaded,
                        "total": event.total,
                        "lengthComputable": event.lengthComputable,
                        "type": event.type,
                        "target": event.target
                    }
                    console.log(JSON.stringify(debugResponse, null, 4));
                }
            }
        };
        xhr.upload.onloadend = function (event) {
            bar.percent = 1;
            bar.draw();
            //document.getElementById("bar").innerHTML = "Terminado";
        };
    }

    // setup XHR Object
    xhr.open(uploadMethod, uploadUrl, uploadType);
    // start response and send data
    xhr.send(data);

}
```

```
}
```

Y por último el código PHP que se encargará de validar y cargar los ficheros. Retornará una serie de datos en formato json para visualizarlos en pantalla.

```
<?php
// debug
error_reporting(E_ALL & ~E_NOTICE);

// configuration
$pass = true;
$error = "";
$upload_dir = "uploads/";
$max_size = 5; // MB
$total_size = $_FILES['upfile']['size']/1024/1024;

// input file values
$inputFile = [
   'name' => $_FILES['upfile']['name'],
   'tmp_name' => $_FILES['upfile']['tmp_name'],
   'size' => $_FILES["upfile"]['size'],
   'type' => $_FILES['upfile']['type'],
   'error' => $_FILES['upfile']['error']
];

// allowed file extensions
$allowed = ["gif", "jpg", "png", "pdf"];
$ext = strtolower(pathinfo($inputFile["name"], PATHINFO_EXTENSION));
if (!in_array($ext, $allowed)) {
   $pass = false;
   $error = 'El tipo de fichero no está permitido.<br />';
   $error .= 'Extensiones permitidas: '.implode(', ',$allowed);
}

// max allowed file size
if ($pass) {
   if ($total_size > $max_size) {
      $pass = false;
      $error = 'El tamaño maximo permitido son '.$max_size.' MB';
   }
```

```
}

// move upload file
if ($pass) {
    $source = $inputFile["tmp_name"];
    $destination = $upload_dir.$inputFile["name"];
    move_uploaded_file($source, $destination);
}

// server json format response
header('Content-Type: application/json');
echo json_encode([
    'status' => $pass,
    'message' => $pass ? 'El fichero '.$inputFile["name"].' ha sido cargado correctamente!
' : $error,
    'debug' => "size: ".$total_size." mb | ".$inputFile["type"]." | name:
".$inputFile["name"]." | error: ".$inputFile["error"],
    'origin' => filter_input(INPUT_SERVER, 'HTTP_ORIGIN')
]);
```

1.4 Cross Origin Resource Sharing (CORS)

Una de las limitaciones iniciales de las comunicaciones AJAX mediante el objeto XHR era que el no poder superar las medidas de seguridad HTTP (CORS) que permiten obtener permiso para acceder a recursos seleccionados desde un servidor, en otro distinto al que este pertenece.

Por razones de seguridad, los exploradores restringen las solicitudes HTTP de origen cruzado iniciadas dentro de un script.

Para evitar esa limitación Microsoft Internet Explorer 8 introdujo el objeto XDomainRequest (XDR) que permitía de forma segura realizar una comunicación cross-domain. El objeto XDR trabajaba muy parecido a XHR con la excepción que solo aceptaba dos parámetros (tipo de petición y URL) ya que sólo se permitía su uso en operaciones asíncronas.

Y aunque el resto de navegadores no permitían conexiones cross-domain se podía usar otro tipo de mecanismos para poder así poder realizar una verificación previa usando cabeceras HTTP y el método setRequestHeader().

1.4.1 La propiedad withCredentials

La propiedad withCredentials nos permite realizar peticiones de forma segura fuera de nuestro dominio cuando realizamos peticiones “asíncronas”.

Le tendremos que pasar un valor booleano que si es true permitirá dicha conexión.

```
var xhr = new XMLHttpRequest();
xhr.open('GET', 'http://www.externaldomain.com/file.php', true);
xhr.withCredentials = true;
xhr.onload = function () {
   console.log(this.responseText);
}
xhr.send();
```

Si por ejemplo la solicitud ha sido realizada desde nuestro dominio http://www.artegrafico.net, el fichero de respuesta podríamos configurarlo de la siguiente forma:

```
<?php
$credentials = true;
$origin = filter_input(INPUT_SERVER, 'HTTP_ORIGIN');

$allowOrigins = array('http://www.artegrafico.net');
if ( !in_array($origin, $allowOrigins) && $credentials ) {
   exit();
}
header("Access-Control-Allow-Origin: " . $origin);
header("Access-Control-Allow-Methods: POST, GET, OPTIONS");
header("Access-Control-Allow-Headers: Origin");
header("Access-Control-Max-Age: 1728000");
if ($credentials) {
   header("Access-Control-Allow-Credentials: true");
}
// Respuesta
header("Content-Type: application/json; charset=utf-8");
```

Bibliografía y Webgrafía

Zakas, N. and Peloquin, J. (2012). Professional JavaScript for Web developers, third edition. Indianapolis, IN: Wrox/Wiley.

Suehring, S. (2013). JavaScript Step by Step. Microsoft Press.

Xhr.spec.whatwg.org. (2019). XMLHttpRequest Standard. [online] Available at: https://xhr.spec.whatwg.org/ [Accessed 11 Mar. 2019].

MDN Web Docs. (2019). XMLHttpRequest. [online] Available at: https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest [Accessed 11 Mar. 2019].

Documentación web de MDN. (2019). Control de acceso HTTP (CORS). [online] Available at: https://developer.mozilla.org/es/docs/Web/HTTP/Access_control_CORS [Accessed 11 Mar. 2019].