

## UD6

# Utilización de mecanismos de comunicación asíncrona

|  |           |
|--|-----------|
| <b>1. Comunicación asíncrona con JavaScript</b>        | <b>1</b>  |
| 1.1. Introducción a AJAX                               | 2         |
| 1.2. Dificultades a la hora de probar el código AJAX   | 3         |
| 1.3. El Objeto XMLHttpRequest                          | 4         |
| 1.3.1. Propiedades                                     | 4         |
| 1.3.2. Métodos   | 4         |
| 1.3.3. Envío y creación de peticiones.                 | 4         |
| 1.3.4. Realizar pruebas básicas.                       | 5         |
| 1.3.5. Comprobar estado de la llamada.                 | 6         |
| 1.4. Envío de parámetros en llamadas a servidor.       | 8         |
| 1.4.1. Enviando datos en la url en llamadas GET.       | 8         |
| 1.4.2. Enviando datos en la cabecera en llamadas POST. | 9         |
| 1.5. Procesamiento de respuestas XML                   | 10        |
| 1.6. Procesamiento de respuestas JSON                  | 11        |
| 1.7. Procesamiento de respuesta HTML                   | 13        |
| <b>2. JSONP Y CORS</b>                                 | <b>14</b> |
| 2.1 JSON con padding (JSONP)                           | 15        |
| <b>3. Enlaces</b>                                      | <b>17</b> |
| <b>4. Resumen de términos</b>                          | <b>17</b> |

## 1. Comunicación asíncrona con JavaScript

En los inicios de internet, cada vez que un usuario realizaba una acción, la página donde se encontraba debía **recargarse completamente** para que refrescara estos cambios (por ejemplo, cuando se añadía un ítem en un carrito electrónico). Esto suponía una **mala experiencia al usuario**, porque había que esperar que la página se descargara de nuevo sin poder interactuar con ellos, por mínimo que fuera el cambio.

Por otra parte, la **carga de trabajo del servidor** también se incrementaba porque tenía que volver a enviar todo el contenido para cada petición que se hacía, y en muchos casos se debían hacer consultas a la base de datos para volver a generar los mismos contenidos que ya había descargado el usuario inicialmente.

## 1.1. Introducción a AJAX

Para resolver el problema de la carga dinámica de datos se utiliza **AJAX** (Asynchronous JavaScript and XML), un conjunto de tecnologías que permiten actualizar los contenidos de una página o aplicación web sin necesidad de recargarla.

Cabe destacar que, aunque originalmente se utilizó XML para el intercambio de datos, hoy en día se utiliza mucho el formato JSON.

### Curiosidad: "Parsear" y "Analizar sintácticamente"

La utilización del término **parsear** es incorrecto pero es posible encontrar materiales que lo usan en lugar de analizar sintácticamente. Hay que tener en cuenta que en ambos casos se hace referencia a la misma operación: interpretar los datos que recibimos desde el servidor.

El objeto **XMLHttpRequest** permite realizar peticiones en segundo plano (son asíncronas), por lo que la aplicación no queda bloqueada cuando se hace la petición, y una vez se recibe la respuesta (con datos o con un mensaje de error) es procesada por la aplicación.

Utilizar esta tecnología también permite crear aplicaciones **single-page**, que consisten en una única página en la que se muestran unos contenidos u otros según las acciones del usuario, sin tener que salir de ella, o las **páginas con desplazamiento vertical ( scroll ) infinito**, en el que una vez se llega al final de la página se cargan más contenidos (como por ejemplo el listado de posts de Facebook o la búsqueda de imágenes de Google).

Un uso muy frecuente es la carga dinámica de los datos de las listas desplegables; por ejemplo, cuando se selecciona una provincia de un desplegable es habitual que se realice una petición al servidor que devuelve el listado de localidades que son añadidas en otra lista. De este modo, se evita tener que descargar todas las localidades cada vez que un usuario visita la página.

### Curiosidad: WebSockets

Cuando tenemos una página web en la que se debe de establecer una conexión parcial o totalmente permanente debemos de recurrir a otro tipo de conexiones con el servidor denominados WebSockets, como pueden ser chats, o actualización en tiempo real de datos, etc. Estos quedan fuera del temario de este módulo pero para aquellos que os interese podéis aprender más en enlaces como:

[https://developer.mozilla.org/es/docs/WebSockets-840092-dup/Writing\\_WebSocket\\_client\\_applications](https://developer.mozilla.org/es/docs/WebSockets-840092-dup/Writing_WebSocket_client_applications)

## 1.2. Dificultades a la hora de probar el código AJAX

Como medida de seguridad, los navegadores aplican **la política del mismo origen** ( same-origin policy ), que **sólo permite realizar peticiones asíncronas dentro del mismo origen**, es decir, tanto la aplicación que realiza la petición como la URL al que envía deben originarse en el mismo dominio. No está permitido hacer

estas peticiones, ni siquiera entre subdominios, por tanto, si una aplicación que se encuentra en [www.example.com](http://www.example.com) hace una petición a [api.example.com](http://api.example.com) el navegador la bloqueará.

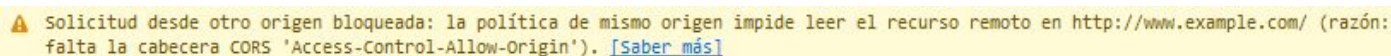
Veamos un ejemplo:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <script>
      let httpRequest = new XMLHttpRequest();
      httpRequest.onreadystatechange = function () {
        console.log("Ha cambiado el estado de la petición.");
      }

      httpRequest.open('GET', 'http://www.example.com', true);
      httpRequest.send();

    </script>
  </head>
  <body>
  </body>
</html>
```

Este código nos lanzará un error con el siguiente formato:

A screenshot of a browser console error message. It features a yellow background and a warning icon (a triangle with an exclamation mark) on the left. The text of the error is: "Solicitud desde otro origen bloqueada: la política de mismo origen impide leer el recurso remoto en http://www.example.com/ (razón: falta la cabecera CORS 'Access-Control-Allow-Origin')." followed by a blue link that says "[Saber más]".

⚠ Solicitud desde otro origen bloqueada: la política de mismo origen impide leer el recurso remoto en http://www.example.com/ (razón: falta la cabecera CORS 'Access-Control-Allow-Origin'). [\[Saber más\]](#)

Cabe destacar que algunos navegadores, como Google Chrome, incluyen el **localhost** dentro de la política del mismo origen y, por tanto, no funcionan correctamente. Por este motivo se recomienda utilizar Mozilla Firefox cuando se desarrollen aplicaciones con fines didácticos.

Es posible evitar esta limitación utilizando otros mecanismos como [CORS](#)( Cross-origin resource sharing ) o JSONP, pero ambas son tecnologías que deben de ser implementadas en el lado del servidor.

### 1.3. El Objeto XMLHttpRequest

El objeto **XMLHttpRequest** forma parte de las especificaciones del W3C y expone una serie de métodos y propiedades que permiten gestionar las llamadas asíncronas y las respuestas obtenidas.

Podemos encontrar una muy buena referencia en la W3Schools en estos 2 enlaces:

- [Tutorial de AJAX](#)
- [XMLHttpRequest](#)

### 1.3.1. Propiedades

Algunas de las propiedades más usadas son:

- **onreadystatechange** : aunque se trata de una propiedad, su función es asignarle una “función callback” que será llamada automáticamente cuando se produzca un cambio en la propiedad **readyState**.
- **readyState** : devuelve el estado de la petición.
- **responseType**: especificamos el formato en el que queremos recibir la respuesta.
- **response**: devuelve la respuesta de la petición tal y como se haya especificado en la propiedad “**responseType**”.
- **responseText** : devuelve la respuesta de la petición como texto.
- **responseXML** : devuelve la respuesta de la petición como XML.
- **withCredentials** ; es una propiedad “booleana” que podemos aplicar antes de realizar una llamada. Si su valor es “true” significa que la llamada al servidor implementa algún tipo de credenciales que el servidor debe validar (normalmente la política Cors). Es un mecanismo de seguridad. Normalmente se implementa añadiendo en el header una serie de parámetros para autenticar nuestra llamada frente al servidor.

### 1.3.2. Métodos

En cuanto a los métodos disponibles para realizar las peticiones básicas están los siguientes:

- **open** : inicializa la petición.
- **overrideMimeType** : permite sobrescribir el tipo multimedia de la respuesta recibida ( text/html, text/plain, application/xml, etc.).
- **send** : envía la petición con los datos pasados como parámetro que pueden ser, entre otros, una cadena de texto, un diccionario de datos o todo el documento. El parámetro es utilizado normalmente en peticiones POST.

Respecto a la detección de eventos , los navegadores modernos admiten el uso del método **addEventListener**, así como el uso de los atajos siguientes:

- **onload** : atajo para añadir un detector del evento load , que se dispara al recibir la respuesta.
- **onerror** : atajo para añadir un detector del evento error , que se dispara si se produce algún error.
- **onprogress** : atajo para añadir un detector del evento progress , que se dispara cuando se actualiza el progreso de la respuesta y permite controlar la proporción recibida respecto al total.

### 1.3.3. Envío y creación de peticiones.

El primer paso es crear el objeto XMLHttpRequest:

```
let httpRequest;  
httpRequest = new XMLHttpRequest();
```

Una vez se ha creado el objeto, a partir de **XMLHttpRequest**, se puede asignar una función al método **onreadystatechange**, que será invocada automáticamente cuando cambia el estado de la petición; es donde se gestiona qué hacer con la respuesta o si se produce un error.

Una vez asignada la propiedad **onreadystatechange** se pueden utilizar los métodos **open y send** para crear la petición y enviarla:

- **open (método, URL , asíncrona):** proporciona la información para crear la petición. El primer parámetro determina el método que se utilizará para la petición ( GET, HEAD, POST, PUT, DELETE, TRACE), el segundo es el URL en el que se realizará la petición, y el tercero (opcional) indica si se quiere hacer la invocación asíncronamente (true) o síncronamente (false), lo que bloquearía la ejecución hasta recibir la respuesta. Este último parámetro es opcional, y si se omite, la petición se hace de manera asíncrona.
- **send (params):** envía la petición previamente preparada con los parámetros especificados. Estos parámetros deben estar en un formato que pueda interpretar el servidor de destino. Si no necesitas enviar parámetros, se deja o bien vacío o con un “null”.

#### 1.3.4. Realizar pruebas básicas.

Para realizar pruebas básicas con AJAX no es necesario tener un servidor funcionando; es posible crear los ficheros de prueba manualmente en una carpeta que sea accesible desde la página (como las imágenes y el código CSS y JS).

Agregue dentro de la misma carpeta de su código un fichero llamado **provincias.xml** con el siguiente contenido:

```
<provincias>
<provincia>Almería</provincia>
<provincia>Granada</provincia>
<provincia>Sevilla</provincia>
<provincia>Malaga</provincia>
</provincias>
```

Además, hay que insertar el siguiente código al ejemplo anterior al principio del punto 1.3.3:

```
httpRequest.onreadystatechange = procesarCambioDeEstado;
httpRequest.open ( 'GET' , 'provincias.xml' , true );
httpRequest.send ( );
function procesarCambioDeEstado( ) {
    console.log( "Ha cambiado el estado de la petición" );
}
```

Como se puede apreciar, una vez se recibe la respuesta invoca la función procesarCambioDeEstado. Es aquí donde se debe implementar la lógica para realizar una acción u otra en función de si se ha finalizado con éxito o se ha producido un error.

#### 1.3.5. Comprobar estado de la llamada.

Lo primero que se debe hacer cuando el estado cambia es comprobar si ya se ha completado o no. El objeto XMLHttpRequest se puede encontrar en 5 estados diferentes y su valor se obtiene a partir de la propiedad “**readyState**”. Los nombres de los valores posibles (y su valor) son los siguientes:

- **Unsent (0).** Aún no se ha abierto la petición con open.

- **Opened (1).** Se ha abierto la petición pero aún no se ha enviado.
- **Headers\_Received (2).** Se ha enviado la petición.
- **Loading (3).** Descargando la respuesta. La propiedad **response** contiene el contenido parcial.
- **Done (4).** Se ha completado la operación.

También es conveniente comprobar el código de la respuesta para comprobar el éxito o no de la operación con la propiedad **"status"**. El valor de esta propiedad será 200 cuando no se ha producido ningún error u otros valores específicos como 404 (página no encontrada) o 500 (error interno del servidor).

Veamos todo esto con un ejemplo completo:

```
let httpRequest = new XMLHttpRequest();
httpRequest.onreadystatechange = procesarCambioEstado;
httpRequest.open ( 'GET' , 'provincias.xml' , true );
httpRequest.send ( null );

function procesarCambioEstado() {

    if (httpRequest.readyState === XMLHttpRequest.DONE) // DONE es igual a 4
    {
        console.log("Estado actual:", httpRequest.readyState);

        if (httpRequest.status === 200) {
            console.log("Se ha devuelto una respuesta correcta. Estado de
respuesta:", httpRequest.status);
            procesarRespuesta(httpRequest.responseText);
        }
        else {
            console.log("Se ha producido un error en obtener la respuesta. Estado de
respuesta: ", httpRequest.status);
        }
    }
    else {
        console.log("No se ha devuelto la respuesta. Estado actual:",
httpRequest.readyState);
    }
}

function procesarRespuesta(resposta) {
    var elementPre = document.createElement('pre');
    elementPre.innerHTML = resposta;
    document.body.appendChild(elementPre);
}
```

### 1.3.6. Haciendo uso de los eventos.

Hay que recordar que cuando se trata de eventos se puede utilizar el método **addEventListener** del objeto XMLHttpRequest o los eventos **onload**, **onerror** y **onprogress** para añadir la función que será llamada al detectarse el evento correspondiente.

**Importante:** Si hacemos uso de estos eventos (onload, onerror, onprogress) podemos dejar de utilizar `httpRequest.onreadystatechange`.

En la mayoría de los casos es recomendable trabajar con los eventos **load** , **error** y **progress** en lugar de controlar los cambios de estado.

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<script>

let httpRequest = new XMLHttpRequest();
httpRequest.onload = procesarRespuesta;
httpRequest.open('GET', 'provincias.xml', true);
httpRequest.send(null);
function procesarRespuesta() {
    var respuesta = httpRequest.responseText;
    var elementPre = document.createElement('pre');
    elementPre.innerHTML = respuesta;
    document.body.appendChild(elementPre);
}
</script>
</head>
<body>
</body>
</html>
```

## 1.4. Envío de parámetros en llamadas a servidor.

Habitualmente, al hacer una petición AJAX hay que añadir uno o más parámetros. Hay que distinguir entre dos tipos de formatos para enviar los parámetros:

- **Cadena de consulta ( query string ):** los parámetros se pasan codificados a la URL , por ejemplo: <http://www.example.com?nom=Maria&cognom=Campmany> .
- **Datos de formulario ( FormData ):** los parámetros se incluyen en la cabecera de la petición.

Cuando el método de envío es **GET** se codificarán los parámetros directamente a la petición (es decir, se trata de una cadena de consulta ), de manera que al llamar el método open del objeto XMLHttpRequest, la URL ya incluye estos parámetros.

En el caso de otros métodos como **POST, PUT/PATCH y DELETE** la petición debe hacerse de una manera diferente, e incluye **modificar la cabecera** de la petición para especificar el formato en el que se envían los datos.

### 1.4.1. Enviando datos en la url en llamadas GET.

Cuando el método de envío es **GET** se codificarán los parámetros directamente a la petición (es decir, se

trata de una cadena de consulta ), de manera que al llamar el método **open** del objeto **XMLHttpRequest**, la URL ya incluye estos parámetros.

```
var httpRequest = new XMLHttpRequest();
var url = 'www.example.com';
var datos= {
    nombre: 'María',
    apellido: 'Campmany Roig'
}

var params = convertirEnParametros(datos);
httpRequest.open('GET', url + params , true);
httpRequest.onload = function() {
    // Aquí es donde se procesa la respuesta de la petición};

httpRequest.send(null);

function convertirEnParametros(datos) {
    var params = '?';
    for (var item in datos) {
        if (params.length > 0) {
            params += '&'
        }

        params += encodeURIComponent(item);
        params += '=';
        params += encodeURIComponent(datos[item]);
    }

    return params;
}
```

Como se puede apreciar, se ha creado una función para convertir el diccionario de datos en una cadena de texto con el formato adecuado para añadirla a la URL . Fíjese que se ha utilizado la función **encodeURIComponent** para convertir los datos (tanto la clave como el valor) en caracteres aceptados por la codificación de la URL , como son los acentos, las tildes y los espacios.

#### 1.4.2. Enviando datos en la cabecera en llamadas POST.

Para añadir el envío de datos como cadena de texto utilizando el método POST no se debe modificar la URL , pero una vez abierta la petición, hay que establecer en la cabecera el tipo de contenido como **“application/x-www-form-urlencoded”**:

```
var httpRequest = new XMLHttpRequest();
var url = 'www.example.com';
var datos = {
    nombre: 'Maria',
    apellidos: 'Campmany Roig'
```



```

}
var params = convertirEnParametros(datos);
httpRequest.open('POST', url, true);
httpRequest.setRequestHeader('Content-type', 'application/x-www-form-urlencoded');
httpRequest.onload = function() {
    // Aquí se procesaría la respuesta de la petición.
};

httpRequest.send(params);

function convertirEnParametros(dades) {
    var params = '';
    for (var dada in dades) {
        if (params.length > 0) {
            params += '&'
        }
        params += encodeURIComponent(dada);
        params += '=';
        params += encodeURIComponent(dades[dada]);
    }
    return params;
}

```

En algunos casos el servicio web puede esperar recibir la información en otro formato. Para ello es necesario especificar este formato a la cabecera y codificar los datos de la manera apropiada. Por ejemplo, si el servidor acepta los parámetros en formato JSON, el código sería el siguiente:

```

var httpRequest = new XMLHttpRequest();
var url = 'www.example.com';
var datos= {
    nombre: 'Maria',
    apellido: 'Campmany Roig'
}
httpRequest.open('POST', url, true);
httpRequest.setRequestHeader('Content-type', 'application/json');

httpRequest.onload = function() {
    // Aquí se procesaría la respuesta de las peticiones
    var respuestaJSON = JSON.parse(httpRequest.responseText);
};

httpRequest.send(JSON.stringify(datos));

```

## 1.5. Procesamiento de respuestas XML

El formato XML es uno de los formatos más utilizados para intercambiar información. Como es el formato por el que, originalmente, se transfería la información utilizando AJAX, el objeto XMLHttpRequest dispone del método responseXML para obtener la respuesta directamente en este formato.

Hay que recordar que las interfaces del DOM se aplican también a los documentos XML, por lo que se pueden utilizar los mismos métodos o propiedades para tratar tanto HTML como XML, tal como puede ver en el ejemplo siguiente. Primeramente, debe crear un archivo llamado provinciasxml.xml con el código XML:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<provincias>
<provincia cp="04">Almeria</provincia>
<provincia cp="17">Granada</provincia>
<provincia cp="25">Jaén</provincia>
<provincia cp="43">Córdoba</provincia>
</provincias>
```

Seguidamente, cree un archivo con el nombre "procesarXML.html" y pegue el código siguiente:

```
<!DOCTYPE html>
<head>
<meta charset="utf-8">
<script>
{
let httpRequest = new XMLHttpRequest();
httpRequest.onload = procesarRespuesta;
httpRequest.open('GET', 'provinciasxml.xml', true)
httpRequest.send(null);

function procesarRespuesta()
{
    var elementoRespuesta = httpRequest.responseXML;
    var provincias = elementoRespuesta.documentElement;
    var elementos = elementoRespuesta.childNodes;
    var lista = document.createElement('ul');

    for (var i = 0; i < elementos.length; i++) {
        if (elementos[i].nodeType == Node.ELEMENT_NODE) {
            var item = processarElement(elementos[i]);
            lista.appendChild(item);
        }
    }

    document.body.appendChild(lista);
}

function processarElement(element) {
    var codiPostal = element.getAttribute('cp');
    var provincia = element.textContent;
    var item = document.createElement('li');
    item.textContent = provincia + ' (CP ' + codiPostal + ')';
    return item;
}
</script>
</head>
```

```
<body>
</body>
```

## 1.6. Procesamiento de respuestas JSON

Para procesar una respuesta en formato JSON (JavaScript Object Notation) utiliza la propiedad **responseText** del objeto **XMLHttpRequest** seguidamente se analiza sintácticamente para convertirla en un objeto de JavaScript, que se puede utilizar como cualquier otro.

Para comprobarlo, cree un archivo de texto llamado provinciasjson.json con el siguiente contenido:

```
{
  "provincias": [
    {"nombre": "Almeria", "cp": "04"},
    {"nombre": "Granada", "cp": "17"},
    {"nombre": "Jaen", "cp": "25"},
    {"nombre": "Cordoba", "cp": "43"}
  ]
}
```

Cabe destacar que el elemento raíz en una respuesta en formato JSON tanto puede ser un objeto como un array .

El código que se utilizará para realizar la petición es prácticamente idéntico al de las peticiones HTML : primero se crea el objeto XMLHttpRequest, seguidamente se envía y finalmente se procesa la respuesta:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Document</title>
</head>
<body>
  <script>
    let httpRequest = new XMLHttpRequest();
    httpRequest.onload = procesarRespuesta;
    httpRequest.open('GET', 'provinciasjson.json', true);
    httpRequest.overrideMimeType('text/plain');
    httpRequest.send(null);

    function procesarRespuesta() {
      var respuesta = JSON.parse(httpRequest.responseText);
      var lista = document.createElement('ul');
      for (var i = 0; i < respuesta.provincias.length; i++) {
        var item = processarElement(respuesta.provincias[i]);
      }
    }
  </script>
</body>
</html>
```

```

        lista.appendChild(item);
    }
    document.body.appendChild(lista);
}

function processarElement(provincia) {
    var item = document.createElement('li');
    item.textContent = provincia.nombre + ' (CP ' + provincia.cp
+ ')';

    return item;
}
</script>

</body>
</html>

```

Como puede ver se ha forzado el tipo multimedia invocando el método **overrideMimeType** con el valor **text/plain** para asegurar que se puede interpretar como texto sin que el navegador lance una excepción.

Fíjese que la implementación en este caso es más simple, ya que la conversión de la respuesta en un objeto de JavaScript es directa y, por tanto, se puede acceder a sus propiedades utilizando la notación de punto (provincia.nom) o la notación de corchetes (provincia["nom"]).

## 1.7. Procesamiento de respuesta HTML

Aunque la interfaz de XMLHttpRequest no incorpora ningún método específico para recuperar la respuesta en formato HTML, es muy fácil hacerlo directamente a partir de la respuesta textual. Primeramente cree un nuevo archivo llamado provincias.html con el siguiente código:

```

<h1>Provincias</h1>
<ul>
<li>Almeria</li>
<li>Granada</li>
<li>Jaen</li>
<li>Malaga</li>
</ul>

```

Como esta estructura no se corresponde con el formato XML válido (debería haber sólo un elemento raíz) y en lugar de enviar la petición al servidor se está usando un archivo local, es necesario sobrescribir el tipo multimedia del archivo de modo que el navegador pueda interpretar la respuesta, ya que por defecto considera que todas las respuestas son de tipo XML.

Para ello se debe invocar el método **overrideMimeType** y pasar el tipo **text/html** como argumento, por lo que el código para realizar una petición simple quedaría así:

```

<html>
<head>

```

```

<meta charset="utf-8">
<script>

    let httpRequest = new XMLHttpRequest();

    httpRequest.onload = procesarRespuesta;
    httpRequest.onprogress = mostrarProgreso;

    function mostrarProgreso(event)
    {
        if (event.lengthComputable) {
            var progreso = 100 * event.loaded / event.total;
            console.log("Completado: " + progreso + "%")
        } else {
            console.log("No se puede calcular el progreso");
        }
    }

    httpRequest.open('GET', 'provincias.html', true);
    httpRequest.overrideMimeType('text/html');
    httpRequest.send(null);

    function procesarRespuesta() {
        var respuesta = httpRequest.responseText;
        var contenedor = document.createElement('div');
        contenedor.innerHTML = respuesta;
        document.body.appendChild(contenedor);
    }
</script>
</head>
<body>
</body>
</html>

```

## 2. JSONP Y CORS

Debido a las limitaciones impuestas por la **política del mismo dominio** no se pueden hacer peticiones AJAX a dominios diferentes de lo que se encuentra la aplicación. Para superar esta limitación hay dos técnicas alternativas: **JSONP (JSON con padding)** y **CORS(Cross-Origin Resource Sharing)**.

Desgraciadamente, ambas requieren que el servidor esté preparado para aceptar estas peticiones de dominios externos y, por consiguiente, no todas las fuentes de datos son accesibles desde aplicaciones web aunque ofrezcan una API (interfaz de programación de aplicaciones) que exponga estos datos.

Hay que tener en cuenta que la política del mismo dominio sólo afecta a los navegadores y, por tanto, los servicios web que no admiten ni JSONP ni CORS siguen siendo accesibles para aplicaciones de escritorio o móviles.

## 2.1 JSON con padding (JSONP) NO ESTUDIAR ESTE APARTADO

La técnica del JSON con padding consiste en cargar la información como si se tratara de un script que incluye una invocación a una función con todos los datos como parámetro (el padding se refiere a esto).

Esta técnica funciona porque la carga de scripts no está sujeta a la política del mismo origen y permite cargar scripts que invoquen automáticamente otras funciones.

Por ejemplo, la respuesta obtenida al cargar el URL:

( [https://api.flickr.com/services/feeds/photos\\_public.gne?jsoncallback=processar&tags=kitten&format=json](https://api.flickr.com/services/feeds/photos_public.gne?jsoncallback=processar&tags=kitten&format=json) )  
incrusta un script en la página con una invocación a la **función processar**, que es el valor del parámetro jsoncallback, y que utiliza el servicio web de Flickr ( [www.flickr.com](http://www.flickr.com) ) para generar la respuesta JSONP y le pasa todos los datos en formato JSON como argumento:

```
procesar({
  "title": "Recent Uploads tagged kitten",
  "link": "http://www.flickr.com/photos/tags/kitten/",
  "description": "",
  "modified": "2016-10-22T18:01:41Z",
  "generator": "http://www.flickr.com/",
  "items": [
    {
      "title": "Cats images Beautiful Eyes via http://ift.tt/29KELz0",
      "link": "http://www.flickr.com/photos/dozhub/29860948003/",
      "media": {"m": "http://farm9.staticflickr.com/8677/29860948003_31626a7d77_m.jpg"},
      "date_taken": "2016-10-22T11:01:41-08:00",
      "description": " <p><a href=\"http://www.flickr.com/people/dozhub/\">dozhub</a>
      posted a photo:</p> <p><a href=\"http://www.flickr.com/photos/dozhub/29860948003/\"
      title=\"Cats images Beautiful Eyes via http://ift.tt/29KELz0\"><img
      src=\"http://farm9.staticflickr.com/8677/29860948003_31626a7d77_m.jpg\"
      width=\"240\" height=\"160\" alt=\"Cats images Beautiful Eyes via
      http://ift.tt/29KELz0\" /></a></p> <p>Cats images Beautiful Eyes -<br /> <br />
      Cats images Beautiful Eyes - Cats, kittens and kittys, cute and adorable! Aww! (via
      <a href=\"http://ift.tt/29KELz0\" rel=\"nofollow\">ift.tt/29KELz0</a><br /> <br />
      - via <a href=\"http://ift.tt/29KELz0\" rel=\"nofollow\">ift.tt/29KELz0</a>. Cats,
      kittens and kittys, cute and adorable! Aww!</p>",
      "published": "2016-10-22T18:01:41Z",
      "author": "nobody@flickr.com (dozhub)",
      "author_id": "143919671@N07",
      "tags": "cat kitty kitten cute funny aww adorable cats"
    }
  ]
});
```

Así pues, una vez se carga el script se ejecuta la función procesar, que previamente habremos definido y recibe como parámetro todos los datos para procesarlas. Como se puede apreciar, sólo se puede utilizar esta técnica si el servidor está preparado para utilizar los datos en este formato.

Véase a continuación un ejemplo que extrae la información del agregador ( feed ) de Flickr para mostrar fotos de gatitos:

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="utf-8">
</head>

<body>

</body>
<script>
  function procesar (datos) {
    var imagenes = datos.items;

    for (var i=0; i<imagenes.length; i++) {
      var img = document.createElement('img');
      img.setAttribute('src', imagenes[i].media.m);
      img.setAttribute('title', imagenes[i].title);
      img.setAttribute('alt', imagenes[i].title);
      document.body.appendChild(img);
    }
  };
</script>

<script
src='http://api.flickr.com/services/feeds/photos_public.gne?jsoncallback=procesar&tags=kitten&format=json'> </script>
</script>
```

## 2.2 Cross-Origin Resource Sharing (CORS)

CORS es un mecanismo que permite (o no permite) a navegadores modernos realizar peticiones AJAX a dominios diferentes de lo que se encuentra la aplicación web, utilizando el objeto XMLHttpRequest pero asegurando la seguridad de la transmisión de datos en colaboración con el servidor.

Al enviar una petición, hay un intercambio de datos entre el cliente y el servidor para comprobar si la transmisión es aceptable o no. Estas comprobaciones pueden incluir diferentes factores como el origen de la petición o la acreditación del usuario.

Si el servidor envía a la cabecera el parámetro **Access-Control-Allow-Origin** con un valor que corresponda al del dominio de la aplicación o “ \* ”, la transmisión se puede llevar a cabo y la respuesta contendrá los datos solicitados. En otros casos es necesario comprobar la acreditación y se necesitará establecer la propiedad **withCredentials=true** del objeto que envía la petición para habilitar el envío de las galletas en el

servidor, donde se comprobará la validez de las credenciales.

Esto se viene haciendo para evitar posibles ataques hackers y la máxima que sigue es la siguiente: <<Un script de un sitio web no puede acceder al contenido de otro sitio >>

Normalmente el **mecanismo CORS se dice que es transparente al código del lado cliente**, ya que es gestionado por la implementación del servidor y los navegadores. Pero esto no es del todo cierto ya que las últimas versiones de navegadores incorporan un mecanismo de seguridad **“Access-Control-Allow-Origin”** por lo que debemos de incluir en las peticiones del lado cliente expresamente de la siguiente forma:

```
fetch(URL, {
  mode: 'cors',
  headers: {
    'Access-Control-Allow-Origin': '*'
  }
  credentials: 'include'
})
.then(response => response.json())
```

En vez de un asterisco también podríamos haber acotado al dominio de nuestra web:

```
Access-Control-Allow-Origin: http://siteA.com
```

Para XMLHttpRequest habilitaremos las credenciales:

```
const xhr = new XMLHttpRequest();
xhr.open('GET', 'https://b.com/c.json', true);
xhr.withCredentials = true;
xhr.send();
```

Pero esto no valdrá de nada si CORS no está bien configurado en el servidor. De otra forma, no nos quedaría otra que traer los datos haciendo uso de algún servidor proxy.

## 2.3. Posibles soluciones para saltarnos restricción CORS.

Como hemos dicho, para que CORS nos dejara hacer llamadas desde localhost a otras APIs, son estas, en el lado servidor las que deberían de habilitar qué dominios son confiables.

Atados de manos, desde el lado cliente podemos hacer una serie de tácticas, trucos para saltarnos esta restricción:

### 1º. Utilizar un servidor proxy intermedio (SOLUCIÓN TEMPORAL (MANOLO & CIA))

Esta táctica es habitual (a modo de prueba, nunca en producción). Podemos usar un servidor “proxy” intermedio el cual redirecciona el tráfico de nuestra petición HTTP hacia nuestro cliente. Un servicio muy famoso y utilizado es el servicio “https://cors-anywhere.herokuapp.com/”.

La forma de utilizarlo sería la siguiente:

```
httpRequest.open( “GET” , https://cors-anywhere.herokuapp.com/http://www.example.com);
```



### 3. Enlaces

- Artículo interesante sobre JSONP:  
<http://albertovilches.com/jsonp-o-la-insercion-dinamica-de-scripts-que-podria-sustituir-a-ajax>
- Artículo interesante sobre el nuevo método Fetch:  
<https://gomakethings.com/why-i-still-use-xhr-instead-of-the-fetch-api/>

### 4. Resumen de términos

**Ajax** - "Asynchronous Javascript and XML". Ajax define un conjunto de tecnologías para ayudar a construir aplicaciones web con una mejor experiencia de usuario debido a que la "actualización" y "refresco" de la pantalla es hecha asincrónicamente usando "javascript" y "xml" (y/o json).

**JSON** - "Javascript Object Notation". JSON es como xml en que este puede ser usado para describir objetos, pero este es más compacto y tiene la ventaja de ser codificado en javascript por lo que un objeto expresado en JSON puede ser convertido dentro un objeto manipulable en javascript.

Por defecto, las peticiones Ajax deben de tener lugar en el mismo dominio de la página donde se origina la petición.

**JSONP** - "**JSON with padding**" - fué creado para permitir realizar peticiones JSON desde diferentes dominios. (**CORS** es una nueva y mejor alternativa a JSONP.)