
AJAX
Asynchronous Javascript And XML


JS
jQuery
write less, do more.

Sumario

AJAX. Asynchronous Javascript and XML....3
1. Introducción....3
2. ¿Para qué sirve AJAX?....4
Modelos de Aplicación Web....4
mi_primer_ajax.html....5
3. El Objeto response....8
4. Ejemplo 2. Ahora fichero JSON, Todavía nada de Servidor… Pero no tardaremos....8
5. Ejemplo 3. Ahora con un servidor Externo. También JSON....10
6. Ejemplo 4. Veamos como funciona de verdad....11
datos.sql....11
base_datos.php....13
cargar_provincias.php....13
cargar_poblacion.php....14
cargar.php....14
7. Opciones de Fetch....17
8. Ejemplo 5. Ajax con POST y mejoras....18
cargar.php....18
javascript.js....18
Pasemos a observar que cambios introduce la petición ajax con feth a la hora de usar el método de envío POST....19
Cambios de POST....19
9. Ejemplo 6. Mi servidor con Json....20
cargar_poblacion.php....20
javascript.js....20

AJAX. Asynchronous Javascript and XML

1. Introducción

AJAX es el acrónimo de **Asynchronous Javascript and XML**, es decir: **Javascript** y **XML Asincrono**. Este acrónimo fue utilizado por primera vez por Jesse James Garret en 2005, en su publicación Ajax: a New Approach to Web Applications si bien los componentes en que se basan y los recursos técnicos de que hace uso ya existían desde muchos años antes.

Normalmente, **AJAX** se define como una técnica para el desarrollo de páginas (sitios) web que implementan aplicaciones interactivas. No obstante, analicemos un poco cada una de las palabras que la forman:

- **Javascript** es un lenguaje de programación conocido por ser interpretado por los navegadores de páginas web.
- **XML** es un lenguaje de descripción de datos pensado fundamentalmente para el intercambio de datos entre aplicaciones, más que entre personas.
- **Asíncrono**: en el contexto de las comunicaciones (y la visualización de una página web no deja de ser una acto de comunicación entre un servidor y un cliente) significa que el emisor emite un mensaje y continúa son su trabajo, dado que no sabe (ni necesita saberlo) cuándo le llegará el mensaje al receptor.

Es decir, que podemos refinar un poco nuestra definición indicando que AJAX es una técnica que permite, mediante programas escritos en Javascript, que un servidor y un navegador intercambien información, posiblemente en XML, de forma asíncrona.

Lo cierto, es que la información intercambiada entre cliente y servidor puede hacerse en múltiples formatos: texto, HTML, JSON y, por supuesto, XML. Además, la comunicación puede ser también síncrona (no es lo habitual), de modo que el cliente espere a que le llegue la información del servidor. En todo caso, los programas que realizamos para lograr este tipo de conexiones se hacen con el lenguaje Javascript.

La historia de AJAX está íntimamente relacionada con un objeto de programación llamado **XMLHttpRequest**. El origen de este objeto se remonta al año 2000, con productos como Exchange 2000, Internet Explorer 5 y Outlook Web Access.

Todo comenzó en 1998, cuando Alex Hopmann y su equipo se encontraban desarrollando la entonces futura versión de Exchange 2000. El punto débil del servidor de correo electrónico era su cliente vía web, llamado OWA Outlook Web Access).

Durante el desarrollo de OWA, se evaluaron dos opciones: un cliente formado sólo por páginas HTML estáticas que se recargaban constantemente y un cliente realizado completamente con HTML dinámico o DHTML. Alex Hopmann pudo ver las dos opciones y se decantó por la basada en DHTML. Sin embargo, para ser realmente útil a esta última le faltaba un componente esencial: "algo" que evitara tener que enviar continuamente los formularios con datos al servidor.

Motivado por las posibilidades futuras de OWA, Alex creó en un solo fin de semana la primera versión de lo que denominó **XMLHTTP**. La primera demostración de las posibilidades de la nueva tecnología fue un éxito, pero faltaba lo más difícil: incluir esa tecnología en el navegador Internet Explorer.

Si el navegador no incluía **XMLHTTP** de forma nativa, el éxito del OWA se habría reducido enormemente. El mayor problema es que faltaban pocas semanas para que se lanzara la última beta de Internet Explorer 5 previa a su lanzamiento final. Gracias a sus contactos en la empresa, Alex consiguió que su tecnología se incluyera en la librería MSXML que incluye Internet Explorer.

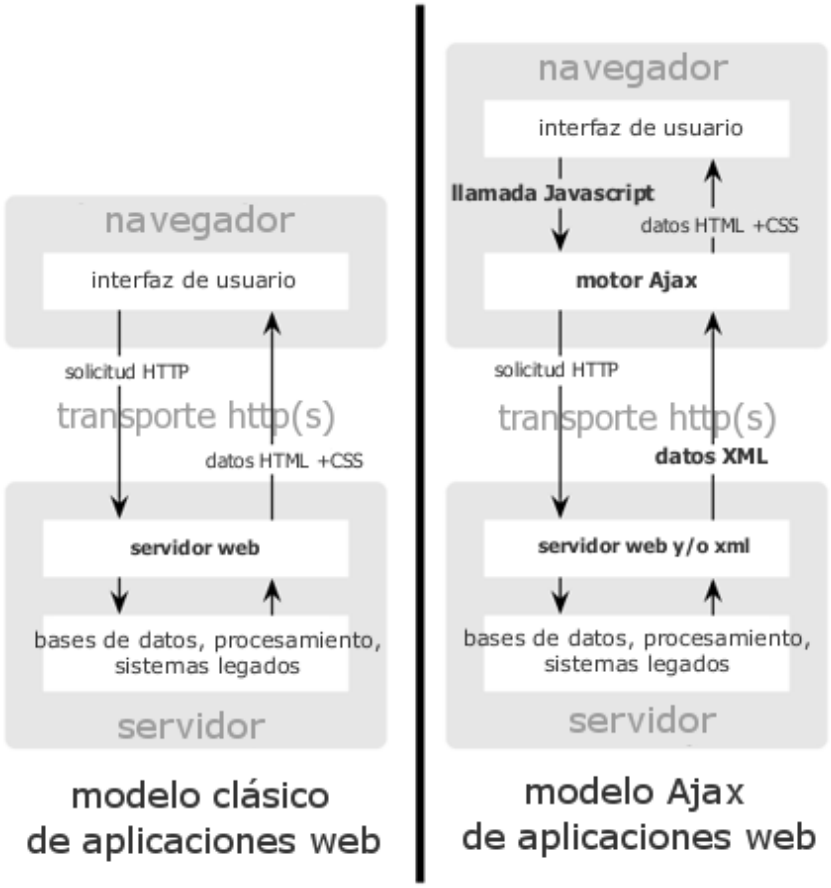
De hecho, el nombre del objeto (**XMLHTTP**) se eligió para tener una buena excusa que justificara su inclusión en la librería XML de Internet Explorer, ya que este objeto está mucho más relacionado con HTTP que con XML.

2. ¿Para qué sirve AJAX?

En esencia, AJAX permite que una página web que ya ha sido cargada solicite nueva información al servidor. Dicho así, no supondría en realidad ningún invento novedoso. Una página web que contiene un enlace permite que se solicite al servidor nueva información cada vez que se pincha dicho enlace. Una página web que contiene un formulario envía información al servidor y recibe de él nueva información, normalmente la respuesta ante los datos que se han enviado. En ambos casos hay una conexión entre el cliente y el servidor.

¿Cuál es la diferencia cuando usamos AJAX? La diferencia es que con AJAX no es necesario recargar toda la página web, como ocurre cuando pinchamos en un enlace o cuando pulsamos el botón submit de un formulario. Con AJAX es posible realizar una conexión a un servidor desde dentro de una página web usando un programa Javascript. Dicho servidor enviará una respuesta; esta respuesta se almacenará en una variable del programa Javascript y, una vez almacenada en la variable, podremos hacer con ella lo que deseemos.

Modelos de Aplicación Web



Comparación de Modelos de Aplicación Web clásico y basado en Ajax Por ejemplo, podemos pedirle al servidor que nos indique qué hora tiene y mostrar dicha hora en el cliente, en una capa dedicada sólo para visualizar este dato. De esta forma, el usuario podría ver la hora correcta que hay en el servidor (posiblemente sincronizada por NTP) y esta sería la misma para todos los usuarios conectados a dicho servidor, sin tener en cuenta la hora que tengan en su ordenador (posiblemente errónea o susceptible de ser modi-

ficada por el usuario). Si actualizamos la hora cada minuto, sin usar AJAX, tendremos que recargar toda la página cada 60 segundos. Sin embargo, con AJAX, simplemente actualizaremos la capa que hemos dedicado a imprimir la hora sin necesidad de alterar el resto de la página.

Pasemos a empezar con ejemplo de ajax

mi_primer_ajax.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">
    <meta name="viewport" content="width=device-width">

    <title>MI PRIMER EJEMPLO</title>
    <link rel="stylesheet" href="style.css">
  </head>

  <body>
    <h1>FETCH LEER FICHERO TEXTO</h1>
    <ul>
      <li><a data-page="page1">Fichero Texto</a></li>
    </ul>
    <article>
    </article>
  </body>

  <script>
    let miTexto = document.querySelector('article');
    let enlace = document.querySelector('ul a');
    enlace.onclick = function(e) {
        e.preventDefault();
        let fichero_Nombre = e.target.getAttribute('data-page');
        VisualizarContenido('fichero1');//AQUÍ ESTÁ LA GRACIA
      };

  function VisualizarContenido(pageId) {
      console.log(pageId);
      let peticion = new Request(pageId + '.txt');//PARAMETROS pagina a
ejecutar
      //montamos  peticion
      fetch(peticion) //ejecutar peticion
      .then(function(respuesta) { //resuelve peticion
        if (!respuesta.ok) { //si la peticion se completa pero falla algo,
p.ej. fichero no existe,
          throw new Error("HTTP error, status = " + respuesta.status);
        }
        return respuesta.text(); //si todo bien devolvemos el la promesa
Resultado
      })
      .then (function (texto){
        miTexto.innerHTML=texto;
      })
      .catch(function(error) { //la petición no se completa, por lo que sea
        miTexto.innerHTML = 'Error: ' + error.message;
      });

    }
  </script>
</html>
```

El funcionamiento es bastante simple: si abrimos nuestro fichero **mi_primer_ajax.html** en el navegador, cargaremos dentro del objeto article el contenido del fichero **fichero1.txt.** Por supuesto el fichero de texto debe existir, claro.

Pasemos a diseccionar poco a poco el código del ejemplo anterior. En primer lugar seleccionamos donde se va a ver el resultado y de donde obtenemos el enlace

```
    let miTexto = document.querySelector('article');
    let enlace = document.querySelector('ul a');
    enlace.onclick = function(e) {
        e.preventDefault();
        let fichero_Nombre = e.target.getAttribute('data-page');
        VisualizarContenido(fichero_Nombre);//AQUÍ ESTÁ LA GRACIA
      };
```

Además asignado al enlace una función que se ejecutará en el caso de que se presiones el click. Seleccionamos el nombre del fichero que vamos a cargar y por último llamamos a la función:
VisualizarContenido()

```
//function VisualizarContenido(pageId) {
        console.log(pageId);
        let peticion = new Request(pageId + '.txt');//PARAMETROS pagina a
ejecutar
        //montamos  peticion
        fetch(peticion) //ejecutar peticion
        .then(function(respuesta) { //resuelve peticion
          if (!respuesta.ok) { //si la peticion se completa pero falla algo,
p.ej. fichero no existe,
            throw new Error("HTTP error, status = " + respuesta.status);
          }
          return respuesta.text(); //si todo bien devolvemos el la promesa
Resultado
        })
        .then (function (texto){
          miTexto.innerHTML=texto;
        })
        .catch(function(error) { //la petición no se completa, por lo que sea
          miTexto.innerHTML = 'Error: ' + error.message;
        });
```

El nombre ya es bastante aclaratorio con respecto a la misión dela función, sirve para abrir el fichero y visualizarlo. ¿Cómo? Generamos una petición a la pagina correspondiente o sea que hace peticiones de información que llegará en formato **XML** (o en texto plano) a través del protocolo **HTTP**.
El objeto, al que hemos llamado ***peticion*** actúa como un “mini-navegador” dentro del cliente en el que se ha cargado nuestra página web . Bueno, actúa más bien como un esclavo al que le damos ordenes básicas, mientras nosotros, el javascript principal hacemos otras cosas, pero es que esta comparación está muy mal vista.

```
 let peticion = new Request(pageId + '.txt');
```

Lo cierto es que la petición no se ejecuta hasta que llamamos al fetch. Seguidamente, y para poder **obtener información del servidor**, abrimos una conexión y le enviamos los datos necesarios para completarla

```
        fetch(peticion) //ejecutar peticion
```

El fetch ejecuta la petición, en este caso, la conexión es de tipo **GET**, intenta recuperar un fichero llamado page1.txt y no necesita muchos más datos, por lo que enviamos la petición sin datos adicionales..

¿Qué ocurre con la petición? fetch es una función asincrónica. Lo que devuelve esta función es un Promise objeto. Este tipo de objeto tiene tres estados posibles: pendiente, cumplido y rechazado. Siempre comienza como pendiente y luego se resuelve o rechaza. Una vez que una promesa se resuelve, ejecuta el then método. Este método toma una función de devolución de llamada como argumento y le pasa el valor resuelto. Echar un vistazo:

```
        .then(function(respuesta) { //resuelve peticion
          if (!respuesta.ok) { //si la peticion falla
            throw new Error("HTTP error, status = " + respuesta.status);
          }
          return respuesta.text(); //si todo bien devolvemos la promesa
        })
```

En resumen si la petición acaba (mal o bien) ejecutará la **function** anónima definida. Pero. ¿qué es la respuesta? Como hemos dicho la respuesta es una promesa, con el resultado de la operación… pero qué es una promesa?
Esa es la clave.
Una promesa es una función que se resolverá a futuro. Cuando realizo una petición puede no resolverse el código, o el código no se ejecuta hasta que se produzca una acción a futuro, por tanto, una promesa es el código que se ejecutará A FUTURO en dicha acción. En nuestro caso, cuando la respuesta se resuelva (cuando ser resuelva la petición asíncrona) tendremos una respuesta, con una serie de datos. Existe muchas manera de definir promesa, en este caso usamos promesas ya predefinidas, por tanto los valores que devuelve están ya predefinidos.

NOTA: El tema de las promesas es altamente interesante en JS, de hecho es una cualidad muy usada, pero se sale un poco del objetivo de estos apuntes. Nos vamos a centrar en el uso de dichas promesas en los casos asíncronos.

Resumiendo, se produce una petición y esta petición se resuelve en respuesta. Dicha respuesta puede indicarnos si todo ha ido bien o mal.

- Mal: levantamos una excepción, que posteriormente resolveremos con catch.
- Bien: Ejecutamos el código que queremos en una función de callback.

Función callback o función de respuesta. Todo bien:

```
        .then (function (texto){ //todo bien
          miTexto.innerHTML=texto;
        })
```

Una vez resuelta la promesa, pasamos a ejecutar el código, la acción, la función de callback que realiza la acción de verdad. Dicha función necesita unos datos de entrada, de hecho la function (texto), lo que recibe es el return repuesta.text (), como entrada. O sea texto=>respuesta.text().
Dentro de la función anónima ejecuto el código con lo que quiero hacer.

Función callback de error. Algo MAL:

```
 .catch(function(error) { //la petición no se completa, por lo que sea
          miTexto.innerHTML = 'Error: ' + error.message;
        });
```

Si algo falla, ejecutamos la excepción y la atrapamos con catch, a la cual le asignamos una función que carga un texto en el texto.

Parece complejo, cierto, pero en realidad, una vez te acostumbras al sistema es bastante claro, y de hecho puede incluso simplificarse el código mucho más, pero esto lo veremos más adelante.

3. El Objeto response

Ya hemos visto que cuando hacemos una petición Fetch, existen dos promesas. La primera es la de la propia petición Fetch, y si esta promesa se resuelve correctamente, obtendremos un objeto de la clase Response/Respuesta. A partir de dicho objeto, solemos usar una segunda promesa para obtener el contenido de la respuesta en el formato deseado.

Las propiedades y métodos más destacables de este objeto Response son:

- **.status** – contiene el código de estado HTTP de la respuesta. Por ejemplo, 200 si todo ha ido bien, 404 si no se ha encontrado el recurso, etc.
- **.statusText** – similar a la propiedad .status pero en este caso el código es una cadena de texto (por ejemplo, para el código 200 el texto es "OK").
- **.ok** – como ya hemos visto, esta propiedad valdrá true si el servidor ha respondido con un código de estado HTTP 2XX, es decir, si la petición AJAX ha sido resuelta satisfactoriamente.
- **.headers** – un objeto con las cabeceras HTTP de la respuesta.
- **.redirected** – propiedad booleana (true/false) que nos indica si la petición ha sido redireccionada.
- **.url** – indica la URL del recurso que nos ha devuelto la respuesta (que por lo general será la misma URL que hemos usado para hacer la petición AJAX, a no ser que se hayan producido redirecciones).
- **.body** – es un objeto que nos da acceso a la respuesta del servidor, pero normalmente usaremos alguno de los siguientes métodos que ya nos proporciona el objeto Response.
- **.text()** – tal y como ya hemos visto, este método nos devuelve una promesa con el contenido de la respuesta del servidor en formato de texto plano.
- **.json()** – de forma similar al anterior, y de nuevo tal y como ya hemos visto, este método nos devuelve una promesa con el contenido en formato JSON.
- **.formData()** – nos devuelve una promesa con el contenido en el mismo formato que se usa para enviar los datos de un formulario HTML.
- **.blob()** – este método nos devuelve una promesa con el contenido en formato binario; puede ser útil cuando el recurso que estamos pidiendo sea de tipo binario, como por ejemplo, una imagen.
- **.arrayBuffer()** – similar al anterior, pero en este caso nos devuelve la información en forma de array.
- **.clone()** – nos permite obtener una copia del objeto. Tan pronto consumimos la promesa asociada al objeto Response con un método como .text() o .json() ya no la podemos volver a consumir, con este método, si lo necesitamos, podemos hacer una copia antes de consumirla.

4. Ejemplo 2. Ahora fichero JSON.

Antes de nada recuerda descargar desde aules la carpeta Ejemplos Fetch, en este caso Fetch con Ficheros.
Veamos el código:

```
    <!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">
    <meta Nombre="viewport" content="width=device-width">
    <title>FETCH CON JSON EJEMPLO</title>

    <link rel="stylesheet" href="style.css">
</head>

  <body>
    <h1>FETCH CON JSON EJEMPLO</h1>
    <ul>
    </ul>

  </body>
  <script>
    let miLista = document.querySelector('ul');

    fetch('productos.json')
    .then(function(respuesta) {
      if (!respuesta.ok) {
        throw new Error("HTTP error, status = " + respuesta.status);
      }
      return respuesta.json();
    })
    .then(function(json) {
      for(let i = 0; i < json.productos.length; i++) {
        let ElementoLi = document.createElement('li');
        ElementoLi.innerHTML = '<strong>' + json.productos[i].Nombre +
'</strong>';
        ElementoLi.innerHTML +=' Lo puedes encontrar en ' +
json.productos[i].Localizacion + '.';
        ElementoLi.innerHTML +=' Coste: <strong>€' + json.productos[i].Precio
+ '</strong>';
        miLista.appendChild(ElementoLi);
      }
    })
    .catch(function(error) {
      let p = document.createElement('p');
      p.appendChild(
        document.createTextNode('Error: ' + error.message)
      );
      document.body.insertBefore(p, miLista);
    });
  </script>
</html>
```

El objetivo de este ejemplo es recalcar lo mismo:

1. Seleccionar Datos de entrada y fichero a cargar. Recuerda descargar el fichero productos.json.
2. Realizar petición, en este caso no generamos el objetos puesto que ejecutamos directamente.
3. La estructura de devolución de información es exactamente la misma, con una diferencia, en este caso devolvemos respuesta.json(). En el ejemplo anterior devuelve sólo texto y en este devuelve el resultado en formato json.
4. Si todo bien: Le llega un array de objetos json, con la siguiente estructura:

a) productos→ ["Nombre", "Precio" , "Localizacion"]
b) Como es un array recorremos dicho array con un for y vamos insertando la información en la lista.

5. Si todo Mal: Ejecutamos el error. Igual que antes.

De momento todos los ejemplo se están ejecutando con GET.

5. Ejemplo 3. Ahora con un servidor Externo. También JSON.

Por fin servidores, pero vaya, no nuestro servidor, este ejemplo me puede servir para iniciarnos en lo que de verdad nos interesa, como recibir datos vía json de un servidor.
Antes de nada recuerda descargar desde aules la carpeta Ejemplos Fetch, en este caso Fetch con Api.
Veamos el código:

pagina.html

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">
    <meta Nombre="viewport" content="width=device-width">
    <title>FETCH CON JSON EJEMPLO</title>
    <link rel="stylesheet" href="style.css">
</head>

<body>
    <h1>FETCH CON JSON EJEMPLO</h1>
    <button id="mostrar">MOSTRAR DATOS </button>
    <ul id="autores"></ul>
    <script src="javascript.js"></script>
</body>
</html>
```

Nada que comentar más allá de que pondremos la información en la etiqueta **ul** cuando pulsemos el botón **mostrar.** El css puedes usar el del ejemplo anterior.

javascript.js

```
function createNode(element) {
  return document.createElement(element);
}

function append(parent, el) {
  return parent.appendChild(el);
}

const boton = document.getElementById("mostrar");
boton.addEventListener("click", mostrarDatos);

function mostrarDatos() {
  const ul = document.getElementById("autores");
  const url = "https://randomuser.me/api/?results=10";

  fetch(url)
    .then((resp) => resp.json())
    .then(function (data) {
      console.log(data);
      let authors = data.results;
      for (let author of authors){
        let li = createNode("li");
        let img = createNode("img");
        let span = createNode("span");
        console.log(img);
        img.src = author.picture.medium;
        span.innerHTML = `${author.name.first} ${author.name.last}`;
        append(li, img);
        append(li, span);
        append(ul, li);
      }
    })
    .catch(function (error) {
```

```
      console.log(error);
    });
}
```

Este ejemplo en esencia es muy parecido al anterior, lo que ocurre es que en vez de realizar el fetch a un archivo lo realizo en una url.
Dicha url, como hacía en el ejemplo del archivo me devuelve un json, el cual es la resolución de la promesa solicita por el fetch.

Hay algunos cambios más con respecto al ejemplo del punto 4, pero son cambios en notación comprimida, en realidad hace lo mismo. Os ponemos la dos notaciones para ver la diferencia. Basicamente en la notación comprimida, obvio un paso, que es el return de la respuesta, el cual sí se hace pero de forma comprimida en el ejemplo 5.

Ejemplo Punto 4.

```
    fetch('productos.json')
    .then(function(respuesta) {
      if (!respuesta.ok) {
        throw new Error("HTTP error, status = " + respuesta.status);
      }
      return respuesta.json();
    })
    .then(function(json) {
   for(let i = 0; i < json.productos.length; i++) {
.....
```

Ejemplo Punto 5.

```
  fetch(url)
    .then((resp) => resp.json())
    .then(function (data) {
   console.log(data);
....
```

Ahondaremos más adelante en la notación comprimida.

Ejercicio 1. Creo que ya lo sospechas… Haz lo mismo con otra API, y tienes para elegir, starwars, Rick&Morty, digimon…

6. Ejemplo 4. Veamos como funciona de verdad

Para ello empecemos con nuestro primer ejemplo con base de datos,evidentemente, este sí llama a nuestro servidor.
Para ello necesitamos los siguientes ficheros

datos.sql

```
SET SQL_MODE="NO_AUTO_VALUE_ON_ZERO";

DROP DATABASE IF EXISTS `ejemplo`;
CREATE DATABASE `ejemplo` DEFAULT CHARACTER SET latin1 COLLATE
latin1_swedish_ci;
USE `ejemplo`;

-- Estructura de tabla para la tabla `localidad`
DROP TABLE IF EXISTS `localidad`;
CREATE TABLE IF NOT EXISTS `localidad` (
  `nombre` varchar(50) NOT NULL DEFAULT '',
  `descripcion_nombre` varchar(50) DEFAULT NULL,
  `fecha_creacion` datetime DEFAULT NULL,
  `extension` int(11) DEFAULT NULL,
  `nombre_provincia` varchar(50) NOT NULL DEFAULT '',
  PRIMARY KEY (`nombre`,`nombre_provincia`),
  KEY `caj_localidad` (`nombre_provincia`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

-- Volcar la base de datos para la tabla `localidad`
INSERT INTO `localidad` (`nombre`, `descripcion_nombre`, `fecha_creacion`,
`extension`, `nombre_provincia`) VALUES
('ASPE', 'IS DIFFERENT', '1924-01-01 00:00:00', 3000, 'ALICANTE'),
('BONETE', NULL, '1950-01-01 00:00:00', NULL, 'ALBACETE'),
('BURRIOL', 'UNA CACA', '1950-01-01 00:00:00', 5000, 'VALENCIA'),
('HELLIN', 'MI CIUDAD', '0000-00-00 00:00:00', 0, 'ALBACETE'),
('HELLIN', 'TU CIUDAD', NULL, NULL, 'MURCIA'),
('LEZUZA', NULL, '1924-01-01 00:00:00', 2000, 'ALBACETE'),
('NOVELDA', 'LA MEJOR', '1925-02-02 00:00:00', 1000, 'ALICANTE'),
('TOTANA', NULL, NULL, 2000, 'MURCIA'),
('VINAROS', 'BUENOS LANGOSTINOS', NULL, 1950, 'CASTELLON'),
('XATIVA', 'QUE BONITA ERES', NULL, 250000, 'VALENCIA');

-- Estructura de tabla para la tabla `provincia`
DROP TABLE IF EXISTS `provincia`;
CREATE TABLE IF NOT EXISTS `provincia` (
  `nombre` varchar(50) NOT NULL DEFAULT '',
  `descripcion_nombre` varchar(50) DEFAULT NULL,
  `fecha_creacion` datetime DEFAULT NULL,
  `extension` int(11) DEFAULT NULL,
  PRIMARY KEY (`nombre`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

-- Volcar la base de datos para la tabla `provincia`
INSERT INTO `provincia` (`nombre`, `descripcion_nombre`, `fecha_creacion`,
`extension`) VALUES
('ALBACETE', NULL, NULL, 200),
('ALICANTE', 'LA MEJOR', '1925-02-02 00:00:00', 1000),
('CASTELLON', 'BUENOS LANGOSTINOS', '1924-01-01 00:00:00', 3000),
('MURCIA', 'QUE BONITA ERES', NULL, 3),
('VALENCIA', 'UNA CACA', '1950-01-01 00:00:00', 5000),
('ZARAGOZA', 'ARRIBA LOS MA?OS', NULL, 5);
```

```
-- Estructura de tabla para la tabla `vendedor`
DROP TABLE IF EXISTS `vendedor`;
CREATE TABLE IF NOT EXISTS `vendedor` (
  `numvendedor` int(11) NOT NULL DEFAULT '0',
  `nombre` varchar(50) DEFAULT NULL,
  `direccion` varchar(50) DEFAULT NULL,
  `sueldo` float NOT NULL,
  `fecha_nacimiento` datetime DEFAULT NULL,
  `cif` varchar(10) DEFAULT NULL,
  PRIMARY KEY (`numvendedor`),
  UNIQUE KEY `cif` (`cif`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

-- Volcar la base de datos para la tabla `vendedor`
INSERT INTO `vendedor` (`numvendedor`, `nombre`, `direccion`, `sueldo`,
`fecha_nacimiento`, `cif`) VALUES
(1, 'HONORATO', NULL, 1000, '2002-02-02 00:00:00', 'B11111111'),
(3, 'HECTOR', 'C/ EL GENERAL', 5000, '2002-02-02 00:00:00', 'B33333333'),
(5, 'FERMIN', 'AVD. MI CASA', 500, '2002-02-02 00:00:00', 'B66666666'),
(6, 'FELIPE', NULL, 1200, '2002-02-02 00:00:00', 'B00000000'),
(7, 'JOSE', NULL, 1000000, NULL, 'B77777777'),
(8, 'ANDRES', 'C/ MICASA', 2000, '2002-02-02 00:00:00', 'B88888888'),
(9, NULL, 'C/ EL GENERAL', 500, NULL, 'B55555555'),
(10, 'SIN NOMBRE', NULL, 250, '2002-02-02 00:00:00', 'C85285285');

-- Estructura de tabla para la tabla `vender`
DROP TABLE IF EXISTS `vender`;
CREATE TABLE IF NOT EXISTS `vender` (
  `numvendedor` int(11) NOT NULL DEFAULT '0',
  `nombre_provincia` varchar(50) NOT NULL DEFAULT '',
  PRIMARY KEY (`numvendedor`,`nombre_provincia`),
  KEY `caj2_vender` (`nombre_provincia`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

-- Volcar la base de datos para la tabla `vender`
INSERT INTO `vender` (`numvendedor`, `nombre_provincia`) VALUES
(1, 'ALBACETE'),
(6, 'ALBACETE'),
(1, 'ALICANTE'),
(10, 'ALICANTE'),
(7, 'MURCIA'),
(9, 'MURCIA'),
(3, 'VALENCIA'),
(8, 'VALENCIA'),
(10, 'VALENCIA'),
(6, 'ZARAGOZA');

-- Claves ajenas para la tabla `localidad`
ALTER TABLE `localidad`
  ADD CONSTRAINT `caj_localidad` FOREIGN KEY (`nombre_provincia`) REFERENCES
`provincia` (`nombre`) ON UPDATE CASCADE;

--  Claves ajena para la tabla `vender`
ALTER TABLE `vender`
  ADD CONSTRAINT `caj1_vender` FOREIGN KEY (`numvendedor`) REFERENCES
`vendedor` (`numvendedor`) ON UPDATE CASCADE,
  ADD CONSTRAINT `caj2_vender` FOREIGN KEY (`nombre_provincia`) REFERENCES
`provincia` (`nombre`) ON UPDATE CASCADE;
```

base_datos.php

```php
<?php
define("HOST","localhost");
define("PASSWORD",""); //Esto dependerá del password claro
define("USUARIO","root");//usamos el root por comodidad, pero es poco seguro
define("BB_DD","ejemplo");//base de datos creada con anterioridad

function conectar(){
$conexion=null;
  try{
      $opciones=  array( PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION );
      $conexion = new PDO('mysql:host='.HOST.';dbname=' . BB_DD,
USUARIO,PASSWORD, $opciones);
  }catch(PDOException $e){
      echo "Ocurrió algo con la base de datos: " . $e->getMessage();
  }
  return $conexion;
}
```

cargar_provincias.php

```php
<?php
include_once("base_datos.php");
$conexion=conectar(); // Se conecta con la base de datos.
 //seleccionamos todos los datos que hay en datos
$sql="SELECT distinct nombre FROM PROVINCIA";
$resultado = $conexion->query($sql);
$provincias = $resultado->fetchAll(PDO::FETCH_ASSOC);

foreach ($provincias as $indice =>$provincia){
      $nombre= $provincia["nombre"];
      echo "<option value=$nombre> $nombre </option>";
}
$conexion=null;
?>
```

Vale cómo código php es una …, pero es un ejemplo rápido

cargar_poblacion.php

```php
<?php
include_once("base_datos.php");
$valor=$_REQUEST["provincia"];

if ($valor=="0") echo "Aqui Van las poblaciones";
else{

      $conexion=conectar(); // Se conecta con la base de datos.
      $sql="SELECT nombre FROM LOCALIDAD WHERE  nombre_provincia='$valor'";
      $resultado = $conexion->query($sql);
      //esto es para generar una espera, NO ES NECESARIO
      sleep(1);
      echo "<SELECT name='poblaciones'>";
      $poblaciones = $resultado->fetchAll();
      foreach ($poblaciones as $indice =>$poblacion){
            $nombre=$poblacion['nombre'];
            echo "<option value='$nombre'> $nombre</option>";
      }
      echo "</SELECT>";
      $conexion=null;
}
?>
```

cargar.php

```
<html>
<head>
</head>
<body>
	<h2>PROVINCIAS Y POBLACIONES</h2>
	<p>Elije una provincia y verás sus poblaciones</p>
	<form action="siguiente.php" id="formulario">
		Provincia:
		<select id='provincia' name='provincia'>
			<option value="0">Elije provincia</option>
			<?php
			include('cargar_provincias.php');
			?>
		</select>
		<div id='pepin'>Aqui Van las poblaciones </div>
		<input type='submit' value='Siguiente' />
	</form>
<script type="text/javascript">
	let desplegable = document.getElementById('provincia');
	let pepin = document.getElementById('pepin');
	desplegable.onchange = function(e) {
		e.preventDefault();
		let provincia = this.value;
		CargarPoblacion(provincia); //AQUÍ ESTÁ LA GRACIA
	};

	function CargarPoblacion(provincia) {
		let myInit = {
			method: 'GET', //GET POST PUT DELETE etc..
			mode: 'cors',
			cache: 'no-cache', // *default, no-cache, reload, force-
cache, only-if-cached
		};
		let peticion = new Request('cargar_poblacion.php?provincia=' +
provincia, myInit);

		/*
		NOTACION COMPRIMIDA
		fetch(peticion)
		    .then(respuesta=>respuesta.text())
		    .then(texto=>(pepin.innerHTML=texto)
		    )
		    .catch(function(error) {
				pepin.innerHTML = "";
			    let p = document.createElement('p');
			    p.appendChild(
				document.createTextNode('Error: ' + error.message)
				);
			    pepin.appendChild(p);
		    });*/

		fetch(peticion)
			.then(function(respuesta) {
				if (!respuesta.ok) {
					throw new Error("HTTP error, status = " +
respuesta.status);
				}
				return respuesta.text();
			})
			.then(function(texto) {
```

```
                        pepin.innerHTML = texto;
                  })
                  .catch(function(error) {
                        let p = document.createElement('p');
                        pepin.innerHTML = "";
                        p.appendChild(
                              document.createTextNode('Error: ' +
error.message)
                        );
                        pepin.appendChild(p);
                  });

      }
</script>
</body>
</html>
```

¿Qué hace el código anterior?
Como diría el tío Jack, vayamos por partes.

```
            <form action="siguiente.php">
                  Provincia:
                  <select id='provincia' name='provincia'>
                        <option value="0">Elije provincia</option>
                  <?php
                   include ('cargar_provincias.php');
             ?>
                  </select >
                  <div id='pepin'>Aqui Van las poblaciones </div>
            <input type='submit' value='Siguiente'/>
        </form>
```

Esto es todo código html y php, lo único es el include de cargar provincias, que rellena con options el desplegable del SELECT.

```
    let desplegable = document.getElementById('provincia');
    let pepin = document.getElementById('pepin');
    desplegable.onchange = function(e) {
        e.preventDefault();
        let provincia = this.value;
        CargarPoblacion(provincia);//AQUÍ ESTÁ LA GRACIA
      };
```

Recogemos los datos de la provincia y obtenemos la capa resultado (es el div pepin...Sí, pepin, un nombre claro y descriptivo XD)
La asignamos el método CargarPoblacion al onchange del desplegable. Por supuesto también podíamos haber usado el **addEventListener**.

```
function CargarPoblacion(provincia) {
    let myInit = {
      method: 'GET', //GET POST PUT DELETE etc..
      mode: 'cors',
      cache: 'no-cache', // *default, no-cache, reload, force-cache, only-if-
cached
    };
    let peticion = new Request('cargar_poblacion.php?provincia='+provincia,
myInit);
```

Lo interesante es el array asociativo myInit, en realidad myInit no es más que los parámetros que le vamos a pasar a la petición. En este caso, le indicamos que utilizaremos un método GET, no usaremos la cache si se repite la petición, modo cors el cual me permite realizar solicitudes a otras webs (info ampliada en el siguiente punto). Vamos que utilizamos myInit para las opciones de la petición.

```
    let peticion = new Request('cargar_poblacion.php?provincia='+provincia,
myInit);
```

Creamos la petición y como es un ejemplo con GET le pasamos vía url el dato provincia.

```
fetch(peticion)
      .then(function(respuesta) {
            if (!respuesta.ok) {
                  throw new Error("HTTP error, status = " + respuesta.status);
                  }
                  return respuesta.text();
            })
            .then(function(texto) {
                  pepin.innerHTML = texto;
            })
            .catch(function(error) {
                  let p = document.createElement('p');
                  pepin.innerHTML = "";
                  p.appendChild(
                        document.createTextNode('Error: ' + error.message)
                  );
                  pepin.appendChild(p);
      });
```

Recogemos la respuesta, que en este caso es texto, y la inyectamos con innerHTML en el div pepin. Si se produce una excepción, en el **catch** añadimos un mensaje de error al contenedor pepin.

NOTA MUY GORDA: Date cuenta que en el cargar_poblacion.php toda la información a devolver está en **echo**, es decir para pasar la información de php a js, se debe imprimir por la salida estándar, en este caso por echo. **Si quiero pasar algo de php a js SIEMPRE USAREMOS echo**.

Por último fíjate que la notación comprimida, la que está comentada, y la notación extensa hacen lo mismo. Lo cierto es que es más habitual encontrarse con la notación comprimida, pero utiliza la que mejor comprendas.

7. Opciones de Fetch

Ya hemos visto que fetch acepta como segundo parámetro un objeto con varias propiedades, el objeto **myInit**. Algunas de las más interesantes son las siguientes:

- **body**: contenido que se envía en la petición al servidor. Puede ser texto, datos de formulario, datos en formato JSON, etc.
- **headers**: aquí podemos indicar una o varias cabeceras HTTP donde podemos indicar información adicional que pueda resultar necesaria para realizar la petición AJAX.
- **cache**: establece la forma en la que funcionará la cache del navegador. El valor por defecto suele funcionar bastante bien, pero si queremos asegurarnos de obtener un resultado fresco, podemos usar el valor no-cache. Si no queremos que el resultado se almacene en la cache del navegador, podemos usar no-store.
- **method**: indica la acción que queremos realizar sobre el recurso. Por lo general usaremos GET para obtener un recurso y POST para enviar información a un recurso; pero el método adecuado por lo general dependerá del servicio Web o API Web al que estemos accediendo.
- **redirect**: existe la posibilidad de que al intentar acceder al recurso el servidor nos devuelva un código de redirección porque el recurso ha cambiado de sitio. En esta propiedad podemos indicar cómo queremos responder ante esta posible situación. Posibles valores son follow si queremos seguir el redireccionamiento, o error si queremos que la promesa sea rechazada
- **credentials**: permite indicar si en la petición AJAX queremos incluir datos de autentificación, cookies, etc.
- **mode**: permite indicar si se consideran válidas peticiones a otros dominios o no. En caso afirmativo la comunicación deberá de seguir el protocolo CORS (Cross-Origin Resource Sharing). Esto tendrá algunas implicaciones como, por ejemplo, limitaciones en el posible valor de la propiedad method (que solo podrá ser HEAD, GET o POST), en el formato de los datos enviados, etc.

8. Ejemplo 5. Ajax con POST y mejoras.

Ahora pasamos a realizar el mismo ejemplo con POST y analizamos las diferencias.
Ten en cuenta que aprovechamos para mejorar el código, y la mayoría de los cambios no tienen tanto que ver con usar un tipo de petición distinta, POST, como con el hecho de mejorar la legibilidad y ordenación del código.

Los ficheros **cargar_provincia.php** y **cargar_poblacion.php** NO SUFREN ningún cambio. Esto es lógico, el servidor debe dar la misma respuesta sea con POST o con GET. En todo caso al recibir datos si se realizaría una distinción usando para recibir los datos el **$_POST** o **$_GET**, pero en este ejemplo al usar **$_REQUEST** no tendríamos que tocar nada.

cargar.php

```
<html>
      <head>
      </head>
      <body>
            <h2>PROVINCIAS Y POBLACIONES</h2>
            <p>Elije una provincia y verás sus poblaciones</p>
            <form action="siguiente.php" id="formulario" METHOD='POST'>
                  Provincia:
                  <select id='provincia' name='provincia'>
                        <option value="0">Elije provincia</option>
                  <?php include ('cargar_provincias.php');  ?>
                  </select >
                  <div id='pepin'>Aqui Van las poblaciones </div>
            <input type='submit' value='Siguiente'/>
        </form>

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

Ahora tenemos un código más corto y limpio, esto es debido a que nos hemos llevado todo el código javascript al fichero javascript.js. Los ficheros cargar_poblacion.php y cargar_provincias.php quedan como estaban.

javascript.js

```
let desplegable = document.getElementById('provincia');
let pepin = document.getElementById('pepin');
desplegable.onchange = function(e) {
    e.preventDefault();
    let provincia = this.value;
    CargarPoblacion(provincia);
};

function CargarPoblacion(provincia) {
    //const datos = new FormData(document.getElementById('formulario'));
    const datos = new FormData();
    datos.append('provincia', provincia);
    const myInit = {
        method: 'POST', //GET POST PUT DELETE etc..
        mode: 'cors',
        cache: 'no-cache', // *default, no-cache, reload, force-cache, only-
if-cached
        body: datos
    };
```

```
    let peticion = new Request('cargar_poblacion.php', myInit);

    //NOTACION COMPRIMIDA
    fetch(peticion)
        .then(respuesta=>respuesta.text())
        .then(texto=>(pepin.innerHTML=texto)
        )
        .catch(function(error) {
            pepin.innerHTML="";
            let p = document.createElement('p');

            p.appendChild(
                document.createTextNode('Error: ' + error.message)
                );
            pepin.appendChild(p);
        });
    }
```

Pasemos a observar que cambios introduce la petición ajax con feth a la hora de usar el método de envío POST.

Cambios de POST

```
    //const datos = new FormData(document.getElementById('formulario'));
    const datos = new FormData();
    datos.append('provincia', provincia);
    const myInit = {
        method: 'POST', //GET POST PUT DELETE etc..
        mode: 'cors',
        cache: 'no-cache', // *default, no-cache, reload, force-cache, only-
if-cached
        body: datos
    };
    let peticion = new Request('cargar_poblacion.php', myInit);
```

1. Generamos un objeto datos, de tipo FormData. Ahí es donde añadiremos con append los distintos elementos a pasar a la petición . El objeto FormData nos crea un objeto formulario. Utilizando el append podemos añadir las variables a pasar al servidor. En código comentado ponemos una alternativa la cual es utilizar el propio formulario para crear el objeto FormData.
2. Evidentemente en el myInit cambiamos el método GET=>POST
3. Usamos la opción body para incluir los datos, que se van a enviar. En este caso incluimos un objeto formData denominado datos.
4. Como ya no lo pasamos con GET, NO hace falta que lo añadamos a la URL.
5. El resto del código se queda como estaba, eso sí esta vez usamos la notación comprimida.

```
    //NOTACION COMPRIMIDA
    fetch(peticion)
        .then(respuesta=>respuesta.text())
        .then(texto=>(pepin.innerHTML=texto)
        )
        .catch(function(error) {
            pepin.innerHTML="";
            let p = document.createElement('p');

            p.appendChild(
                document.createTextNode('Error: ' + error.message)
                );
            pepin.appendChild(p);
        });
    }
```

9. Ejemplo 6. Mi servidor con Json.

En realidad esto ya lo hemos visto en el punto 5 con el ejemplo 3. Es más, lo hemos hecho varias veces con las api de starwars o de Rick y Morty, pero nunca hasta ahora lo hemos hecho con un ejemplo en un servidor propio. Vayamos a ello.
Eso sí podemos aprovechar mucho código, no hace falta tocar los ficheros

- base_datos.php
- cargar_provincias.php
- cargar.php

Por tanto pasemos a modificar los archivos correspondients.

cargar_poblacion.php

```
<?php
include_once("base_datos.php");
$valor=($_REQUEST["provincia"])??"";

if ( $valor=="0") echo json_encode([]);
else{

      $conexion=conectar();
      $sql="SELECT * FROM LOCALIDAD WHERE  nombre_provincia='$valor'";
      $resultado = $conexion->query($sql);
      $poblaciones = $resultado->fetchAll(PDO::FETCH_ASSOC);
      $conexion=null;
      echo json_encode($poblaciones);
}
?>
```

Los cambios son mínimos, pero fijate que en este caso no devolvemos texto sino que en el echo colocamos json_encode, el cual enviará a js texto en formato json. Además por cambiar no sólo cogemos el nombre sino todos los campos de la tabla.

javascript.js

```
function createNode(element) {
  return document.createElement(element);
}

function append(parent, el) {
  return parent.appendChild(el);
}

let desplegable = document.getElementById("provincia");
let pepin = document.getElementById("pepin");
desplegable.onchange = function (e) {
  e.preventDefault();
  let provincia = this.value;
  CargarPoblacion(provincia);
};

function CargarPoblacion(provincia) {
  //const datos = new FormData(document.getElementById('formulario'));
  const datos = new FormData();
  datos.append("provincia", provincia);
  const myInit = {
    method: "POST", //GET POST PUT DELETE etc..
    mode: "cors",
    cache: "no-cache", // *default, no-cache, reload, force-cache, only-if-
```

```
cached
    body: datos,
  };
  let peticion = new Request("cargar_poblacion.php", myInit);

  //NOTACION COMPRIMIDA
  fetch(peticion)
    .then((resp) => resp.json())
    .then(function (provincias) {
      pepin.innerHTML="";
      //console.log(provincias);
      if (provincias.length>0){
        let select = createNode("select");
        select.id = "poblaciones";
        select.name = "poblaciones";
        for (let p of provincias) {
            let option = createNode("option");
            option.text=p.nombre;
            append(select, option);
        }
        append(pepin, select);
        }
        else{
            pepin.innerHTML="Aqui Van las poblaciones";
        }
    })
    .catch(function(error) {
        pepin.innerHTML="";
        let p = document.createElement('p');
        p.appendChild(
            document.createTextNode('Error: ' + error.message)
            );
        append(pepin,p);
    });
}
```

La clave está aquí:

```
 //NOTACION COMPRIMIDA
  fetch(peticion)
    .then((resp) => resp.json())
    .then(function (provincias) {
      pepin.innerHTML="";
      //console.log(provincias);
      if (provincias.length>0){
        let select = createNode("select");
        select.id = "poblaciones";
        select.name = "poblaciones";
        for (let p of provincias) {
            let option = createNode("option");
            option.text=p.nombre;
            append(select, option);
        }
        append(pepin, select);
        }
```

A diferencia de antes que recibimos texto “puro” aquí recibimos un texto que respresenta json, por tanto resp.json() genera automáticamente el conjunto de objetos json con las provincias. Una vez tengo los objetos, ya está recorro y monto lo que quiera.