UD07 - APIs en HTML5 y usos avanzados

1. Introducción

En este tema varemos diferentes APIs incluidas en HTML5 (como la de Local Storage) y otras que se han hecho muy populares como la de Google Maps.

HTML5 incluye un buen número de APIs que facilitan el trabajo con cosas complejas, como

- APIs para manejo de audio y vídeo
- APIs para manejo de formularios
- API de dibujo canvas (en el módulo de DIW)

Aquí comentaremos Storage, Drag&Drop, Geolocation, File Access, Communication, Web Workers, History y Offline

2. HTML Drag And Drop API

Con HTML5 es muy sencillo arrastrar y soltar elementos en una página web. Podemos arrastrar y soltar cualquier nodo DOM (una imagen, un archivo, enlaces, texto seleccionado, ...). Para ello sólo es necesario que ese elemento tenga el atributo `dragable="true"`. Si le ponemos `false` no se podrá arrastrar y si no definimos el atributo podrá o no arrastrarse según el valor predeterminado del navegador (en la mayoría son *dragables* las imágenes, los links y las selecciones de texto).

Al arrastrar y soltar intervienen 2 elementos diferentes:

- el elemento que estamos arrastrando
- el elemento sobre el cual lo soltamos

Para poder realizar la operación *event* tiene una propiedad llamada **dataTransfer** que es un objeto en el que almacenamos qué elemento estamos arrastrando (o cualquier otra cosa que queramos) y así cuando se suelte sobre el elemento destino éste último pueda saber quién se le ha soltado.

Los pasos para arrastrar y soltar un elemento son:

1. El elemento debe ser ***draggable***
2. Capturamos el evento **dragstart**. Este evento se produce sobre un elemento cuando comenzamos a arrastrarlo. Deberemos almacenar en el *dataTransfer* quién está siendo arrastrado (si no guardamos nada se guarda automáticamente su `src` si es una imagen o su `href` si es un enlace). Indicaremos el tipo del dato que estamos almacenando (texto plano, HTML, fichero, etc) y su valor. Ej.:

```
<img id="imgGoogle"
src="https://upload.wikimedia.org/wikipedia/commons/thumb/5/51/Google.png/320px-Google.png">
<div id="zonaDrop" class="drop"></div>
```

```
document.getElementById('imgGoogle').addEventListener('dragstart', (event) => {
    event.dataTransfer.setData('text/plain', event.target.id);  // Estamos guardando el texto 'imgGoogle'
})
```

3. Capturamos el evento **dragover**. Este evento se produce cada pocas décimas de segundo sobre elemento sobre el que se está arrastrando algo. Por defecto no se puede soltar un elemento en ningún sitio así que capturamos este evento para evitar que el navegador haga la acción por defecto e impida que se suelte lo que estamos arrastrando. Ej.:

```
document.getElementById('zonaDrop').addEventListener('dragover', (event) => {
    event.preventDefault();
})
```

4. Capturamos el evento **drop**. Este evento se produce sobre elemento sobre el que se suelta lo que estábamos arrastrando. Lo que haremos es evitar el comportamiento por defecto del navegador (que en caso de imágenes o enlaces es cargarlos en la página), obtener quién se ha soltado a partir del objeto *dataTransfer* y realizar lo que queramos, que normalmente será añadir el objeto arrastrado como hijo del objeto sobre el que se ha hecho el *drop*. Ej.:

```
document.getElementById('zonaDrop').addEventListener('drop', (event) => {
    event.preventDefault();
    const data=event.dataTransfer.getData("text/plain");       // Obtenemos ìmgGoogle'
    event.target.appendChild(document.getElementById(data));
})
```

NOTA: si hacemos *draggable* un elemento, por ejemplo un párrafo, ya no se puede seleccionar con el ratón ya que al pinchar y arrastrar se mueve, no se selecciona. Para poder seleccionarlo debemos pinchar y arrastrar el ratón con la tecla *Alt* pulsada o hacerlo con el teclado.

Podemos obtener más información de esta API MDN web docs.

> **PRUEBA 1_UD7**: mira y modifica el ejemplo de w3schools para entender bien el funcionamiento del Drag&Drop (ten en cuenta que en vez de .addEventListener() las llamadas a los escuchadores están puestas como atributos del HTML pero el funcionamiento es el mismo).

3. Almacenamiento en el cliente: API Storage

Antes de HTML5 la única manera que tenían los programadores de guardar algo en el navegador del cliente (como sus preferencias, su idioma predeterminado para nuestra web, etc) era utilizando *cookies*. Las cookies tienen muchas limitaciones y es engorroso trabajar con ellas.

HTML5 incorpora la API de Local Storage para subsanar esto. Además existen otros métodos de almacenamiento en el cliente más avanzados como IndexedDB (es un estándar del W3C).

El funcionamiento de la API Storage es muy sencillo: si el navegador la soporta dentro del objeto *window* tendremos los objetos **localStorage** y **sessionStorage** donde podremos almacenar información en el espacio de almacenamiento local (5 MB o 10 MB por sitio web según el navegador, que es mucho más de lo que teníamos con las cookies). La principal diferencia entre ellos es que la información almacenada en localStorage nunca expira, permanece allí hasta que la borremos (nosotros o el usuario) mientras que la almacenada en sessionStorage se elimina automáticamente al cerrar la sesión el usuario.

Mediante Javascript puedo saber si el navegador soporta o no esta API simplemente mirando su `typeof`:

```
if (typeof(Storage) === "undefined")     // NO está soportado
```

Tanto localStorage como sessionStorage son como un objeto global al que tengo acceso desde el código. Lo que puedo hacer con ellos es:

- Guardar un dato: `localStorage.setItem("dato", "valor")` o también `localStorage.dato = "valor"`
- Recuperar un dato: `let miDato=localStorage.getItem("dato")` o también `let miDato = localStorage.dato`
- Borrar datos: `localStorage.removeItem("dato")` para borrar 'dato'. Si quiero borrar TODO lo que tengo `localStorage.clear()`
- Guardar un objetos (o arrays): lo guardamos en JSON, `localStorage.setItem("dato", JSON.stringify("objeto"))`
- Recuperar el objeto: `ler miObjeto = JSON.parse(localStorage.getItem("dato"))`
- Cómo saber cuántos datos tenemos: `localStorage.length`

Cada vez que cambia la información que tenemos en nuestro localStorage se produce el evento **storage**. Si, por ejemplo, queremos que una ventana actualice su información si otra cambia algún dato del storage haremos:

```
window.addEventListener("storage", actualizaDatos);
```

y la función 'actualizaDatos' podrá leer de nuevo lo que hay y actuar en consecuencia.

> **PRUEBA 2_UD7**: comprueba qué tienes almacenado en el localStorage y el sessionStorage de tu navegador. GUarda y recupera algunas variables. Luego cierra el navegador y vuelve a abrir la página. ¿Están las variables guardadas en localStorage? ¿Y las de sessionStorage?

Puedes ver un ejemplo en este vídeo de cómo almacenar en el *Storage* datos del usuario.

3.1 A tener en cuenta

localStorage, *sessionStorage* y *cookies* almacenan información en un navegador específico del cliente, y por tanto:

- No podemos asegurar que permanecen ahí
- Puede ser borrada/manipulada
- Puede ser leída, por lo que no adecuada para almacenar información sensible pero sí para preferencias del usuario, marcadores de juegos, etc

Podríamos usar localStorage para almacenar localmente los datos con los que trabaja una aplicación web. Así conseguiríamos minimizan los accesos al servidor y que la velocidad de la aplicación será mucho mayor al trabajar con datos locales. Pero periódicamente debemos sincronizar la información con el servidor.

3.2 Storage vs cookies

Ventajas de localStorage:

- 5 o 10 MB de almacenamiento frente a 4 KB de las cookies
- Todas las cookies del dominio se envían al servidor con cada petición al mismo lo que aumenta el tráfico innecesariamente

Ventajas de las cookies:

- Soportadas por navegadores antiguos
- Las cookies ofrecen algo de protección frente a XSS (Cross-Site Scripting)/Script injection

3.3 Cookies

Son pequeños ficheros de texto y tienen las siguientes limitaciones:

- Máximo 300 cookies, si hay más se borran las antiguas
- Máximo 4 KB por cookie, si nos pasamos se truncará
- Máximo 20 cookies por dominio

Cada cookie puede almacenar los siguientes datos:

- Nombre de la cookie (obligatorio)
- Valor de la misma
- expires: timestamp en que se borrará (si no pone nada se borra al salir del dominio)
- max-age: en lugar de *expires* podemos indicar aquí los segundos que durará la cookie antes de expirar
- path: ruta desde dónde es accesible (/: todo el dominio, /xxx: esa carpeta y subcarpetas). Si no se pone nada sólo lo será desde la carpeta actual
- domain: dominio desde el que es accesible. Si no ponemos nada lo será desde este dominio y sus subdominios
- secure: si aparece indica que sólo se enviará esta cookie con https

Un ejemplo de cookie sería:

```
username=John Doe; expires=Thu, 18 Dec 2013 12:00:00 UTC;
```

Se puede acceder a las cookies desde **document.cookie** que es una cadena con las cookies de nuestras páginas. Para trabajar con ellas conviene que creemos funciones para guardar, leer o borrar cookies, por ejemplo:

- Crear una nueva cookie

```
function setCookie(cname, cvalue, cexpires, cpath, cdomain, csecure) {
  document.cookie = cname + "=" + cvalue +
    (cexpires?";expires="+cexpires.toUTCString():"") +
    (cpath?";path="+cpath:"") +
    (cdomain?";domain="+cdomain:"") +
    (csecure?";secure":"")
}
```

- Leer una cookie

```
function getCookie(cname) {
    if(document.cookie.length>0){
        start = document.cookie.indexOf(cname + "=")
        if (start!=-1) {   // Existe la cookie, busquemos dónde acaba su valor
            //El inicio de la cookie, el nombre de la cookie mas les simbolo '='
            start = start + nombre.length + 1
            //Buscamos el final de la cookie (es el simbolo ';')
            end = document.cookie.indexOf(";",start + cname.length + 1)
            if (end === -1) {   // si no encuentra el ; es que es la última cookie
                end=document.cookie.length;
            }
            return document.cookie.substring(start + cname.length + 1, end))
        }
    }
    return ""   // Si estamos aquí es que no hemos encontrado la cookie
}
```

- Borrar una cookie

```
function delCookie(cname) {
    return document.cookie = cname + "=;expires=Thu, 01 Jan 1970 00:00:01 GMT;"
}
```

Podéis ver en este vídeo un ejemplo de cómo trabajar con cookies, aunque como ya hemos dicho lo recomendable es trabajar con *Storage*.

4. Geolocation API

Esta API permite a la aplicación web acceder a la localización del usuario si éste da su permiso. Muchos navegadores sólo permiten usarlo en páginas seguras (https).

Podemos acceder a esta API mediante el objeto **geolocation** de *navigator*. Para saber si nuestro navegador soporta o no la API podemos hacer:

```
if (geolocation in navigator)   // devuelve true si está soportado
```

Para obtener la posición este objeto proporciona el método **navigator.geolocation.getCurrentPosition()** que hace una petición **asíncrona**. Cuando se reciba la posición se ejecutará la función *callback* que pasemos como parámetro y que recibirá las coordenadas de la localización. Podemos pasar otra como segundo parámetro que se ejecutará si se produce algú error y que recibirá un objeto con la propiedad *code* que indica el error producido. Ej.:

```
navigator.geolocation.getCurrentPosition(
  (position) => {
    pinta_posicion(position.coords.latitude, position.coords.longitude)
  },
  (error) => {
    switch(error.code) {
      case error.PERMISSION_DENIED: // El usuario no autoriza al navegador a acceder a la localización
        msg = "El usuario ha denegado la petición de geolocalización"
        break
      case error.POSITION_UNAVAILABLE: // No se puede obtener la localización
        msg = "La información de localización no está disponible."
        break
      case error.TIMEOUT: // Ha expirado el tiempo para obtener la localización
        msg = "Ha expirado el tiempo para obtener la localización"
        break
      case error.UNKNOWN_ERROR:
        msg = "Se ha producido un error desconocido."
        break
    }
    muestra_error(msg)
```

```
    }
)
```

Si queremos ir obteniendo continuamente la posición podemos usar el método **.watchPosition()** que tiene los mismos parámetros y funciona igual pero se ejecuta repetidamente. Este método devuelve un identificador para que lo podemos detener cuando queremos con **.clearWatch(ident)**. Ej.:

```
const watchIdent = navigator.geolocation.watchPosition(
  (position) => pinta_posicion(position.coords.latitude,
position.coords.longitude),
  (error) => muestra_error(error)
)
...
// Cuando queremos dejar de obtener la posición haremos
navigator.geolocation.clearWatch(watchIdent)
```

Las principales propiedades del objeto de localización (algunas sólo estarán disponible cuando usemos un GPS) son:

- coords.latitude: latitud
- coords.longitude: longitud
- coords.accuracy: precisión (en metros)
- coords.altitude: altitud (en metros, sobre el nivel del mar)
- coords.altitudeAccuracy: precisión de la altitud
- coords.heading: orientación (en grados)
- coords.speed: velocidad (en metros/segundo)
- timestamp: tiempo de respuesta UNIX

Podemos pasarle como tercer parámetro al método getCurrentPosition un objeto JSON con una o más de estas propiedades:

- enableHighAccuracy: true/false que indica si el dispositivo debe usar todo lo posible para obtener la posición con mayor precisión (por defecto false porque consume más batería y tiempo)
- timeout: milisegundos a esperar para obtener la posición antes de lanzar un error (por defecto es 0, espera indefinidamente)
- maximumAge: milisegundos que guarda la última posición en caché. Si se solicita una nueva posición antes de expirar el
  tiempo, el navegador devuelve directamente el dato almacenado

Podemos obtener más información de esta API en MDN web docs y ver y modificar ejemplos en w3schools y muchas otras páginas. i

5. Google Maps API

Para poder utilizar la API en primer lugar debemos obtener una API KEY de Google.

Una vez hecho para incluir un mapa en nuestra web debemos cargar la librería para lo que incluiremos en nuestro código el script:

```
<script async defer
  src="https://maps.googleapis.com/maps/api/js?
key=ESCRIBE_AQUI_TU_API_KEY&callback=initMap">
</script>
```

(el parámetro callback será el encargado de llamar a la función initMap() que inicie el mapa)

Ahora incluir un mapa es tán sencillo como crear un nuevo objeto de tipo *Map* que recibe el elemento DOM donde se pintará (un div normalmente) y un objeto con los parámetros del mapa (como mínimo su centro y el zoom):

```
let map
function initMap() {
  map = new google.maps.Map(document.getElementById('map'), {
    center: {lat: 38.6909085, lng: -0.4963000000000193},
    zoom: 12
  })
}
```

Por su parte añadir un marcador es igual de simple. Creamos una instancia de la clase *Marker* a la que le pasamos al menos la posición, el mapa en que se creará y un título para el marcador:

```
let marker = new google.maps.Marker({
  position: {lat: 38.6909085, lng: -0.4963000000000193},
  map: map,
  title: 'IES Mestre Ramon Esteve'
})
```

Aquí tenéis el ejemplo anterior:

Podemos obtener más información de esta API en Google Maps Plataform, en el tutorial de w3schools y en muchas otras páginas.

6. Más cosas a aprender en Javascript

Hace unos años Javscript era considerado un lenguaje de programación de segunda categoría, que se usaba para hacer molestas páginas web. Hoy en día el navegador es la aplicación más importante de un equipo y con él, además de navegar, se ejecutan todo tipo de aplicaciones. Además HTML5 y JavaScript han pasado de estar solo en nuestro navegador a ser un pilar básico de las plataformas móviles, de aplicaciones de escritorio e incluso JavaScript lo encontramos en servidores (con Node.js) o como lenguaje estándar de algunos entornos de escritorio (como GNOME para Linux).

Por ello HTML5 y Javascript siguen su contínuo crecimiento... y su contínua evolución que les permite hacer cada vez más cosas. En esta página vamos a hablar muy brevemente de algunas de las características que están incorporando:

- Web Components
- Web Sockets
- Web Workers
- Typescript
- ...

6.1 WebComponents

Son distintas tecnologías que podemos usar (todas o alguna de ellas) para crear componentes reutilizables para nuestras páginas HTML. Las tecnologías que hay tras los Web Components son:

- Custom Elements: permite crear elementos HTML personalizados, es decir, nuevas etiquetas definidas por nosotros con funcionalidad propia. Por ejemplo

```
<comp-calendar
    mode="month"
    date="2020-02-23"
    on-select="dateSelected()" >
</comp-calendar>
```

- HTML Templates: la etiqueta `<template>` permite definir fragmentos de código HTML que no serán renderizados y que usaremos más adelante. Pueden incluir **slots** o huecos a los que se pasa un contenido
- Shadow DOM: permite asociar un DOM oculto a un elemento. Esto permite que tenga su propio código JS y estilos CSS aislados del resto del DOM
- ES Modules: Es el estándar de ECMAScript para importar módulos Javascript.

Un WebComponent es un elemento que creamos y que tiene su propia representación (HTML) y funcionalidad (establecida con Javascript). Este elemento es reusable y compartible y se contruye sin librerías externas, sólo con HTML5, ES6 y CSS3.

Algunos ejemplos de componentes útiles que podríamos usar son:

- componente para loguearnos mediante Google, Facebook, etc
- componente que me muestre el tiempo en una ciudad
- componente para hacer un modal
- ...

Hay infinidad de páginas donde podemos aprender más sobre WebComponents y cómo crear nuestro propio componente, como:

- Web Components | MDN
- Introduction - webcomponents.org
- ¿Qué son los WebComponents? - Javascript en español
- Carlos Azaustre - Cómo crear un WebComponent de forma nativa
- ...

En resumen debemos crear un fichero donde definimos la clase de nuestro componente que debe heredar de `HTMLElement`. Es conveniente que su nombre (y por tanto el de la etiqueta que usaremos para mostrarlo) tenga al menos 2 palabras para evitar que pueda entrar en conflicto con posibles futuras etiquetas de HTML (por ejemplo podría ser <social-login> o <my-weather>). En esta clase definiremos el HTML y el estilo que tendrá nuestro componente, así como su comportamiento.

Actualmente no todos los navegadores ofrecen soporte para WebComponents. Esto junto al hecho de que frameworks como Vue, Angular o React ofrecen soluciones con sus propios componentes han hecho que el uso de los WebComponents no acabe de despegar. A pesar de ello hay lugares como WebComponents.org donde podemos encontrar un catálogo de componentes hechos y que podemos usar en nuestras páginas.

Por su parte Google ha desarrollado la librería Polymer para ayudarnos a crear nuestros propios componentes basados en WebComponents y los principales frameworks JS como Angular o Vue permiten crear sus propios componentes de forma muy sencilla, com veremos en el bloque de Vue.

6.2 WebSockets

WebSockets es una tecnología basada en el protocolo **ws** que permite establecer una conexión continua *full-duplex* entre un cliente (puede ser un navegador) y un servidor. La conexión siempre la abre el cliente pero una vez abierta no se cierra por lo que el servidor puede comunicar en cualquier momento con el cliente y enviarle información.

Ejemplo:

```
let exampleSocket=new WebSocket(uri);
exampleWebsocjet.onopen=function(event) {
    console.log('Se ha establecido la conexión');
}
exampleSocket.onclose=function(event) {
    console.log('Se ha cerrado la conexión');
}
exampleSocket.onerror=function(event) {
    console.log('Se ha producido un error en la conexión');
}
exampleSocket.onmessage=function(event) {
    console.log('Se ha recibido el mensaje:' + event.data);
}
```

El *uri* de la conexión deberá usar el protocolo **ws** (o wss), no http (ej. "ws://miservidor.com/socketserver"). El evento *open* se produce cuando la propiedad *readyState* cambia a OPEN y el *close* cuando cambia su valor a CLOSED. Cada vez que se reciba algo del servidor se produce el evento *message* y en la propiedad **data** del mismo tendremos lo que se nos ha enviado.

Para enviar algo al servidor usamos el mátodo **.send**. Lo que le enviamos ex texto en formato utf-8 (o un objeto convertido a JSON):

```
exampleSocket.send('Your message');
exampleSocket.send(JSON.stringify(msg));
```

También podemos enviar (y recibir) imágenes (convertidas a ArrayBuffer) o ficheros como un objeto Blob.

Para cerrar la conexión llamamos al método **.close()**:

```
exampleSocket.close();
console.log('Conexión cerrada');
```

Para programar la parte del servidor podemos usar librerías que nos ayudan como PHP-WebSockets, SocketIO, ...

Las aplicaciones de esta tecnología son muchas:

- Juegos multjugador
- Aplicaciones de chat

- Actualización en tiempo real de cotizaciones de bolsa, recursos en uso o cualquier otra información
- ...

Podemos practicar con www.websocket.org que tiene un servidor websocket que devuelve lo que le enviamos. En esta web también tenemos ejemplos de aplicaciones.

Saber más:

- MDN: Escribiendo aplicaciones con WebSockets
- WebSocket - El Tutorial de JavaScript Moderno
- Carlos Azaustre: Crear chat con WebSockets

6.3 WebWorkers

Permite ejecutar scripts en hilos separados que se ejecutan en segundo plano y se comunican con la tarea que los crea mediante el envío de mensajes.

Cuando se está ejecutando un script la página no responde hasta que finaliza su ejecución. Si el script lo ejecuta un web worker la página será funcional (podemos interactuar con ella) ya que le ejecución del script se realiza en segundo plano en otro hilo.

Por ejemplo, crearemos un fichero **worker_count.js** con el código:

```
var i = 0;

function timedCount() {
  i = i + 1;
  postMessage(i);
  setTimeout("timedCount()",500);
}

timedCount();
```

La función *postMessage* envía un mensaje a la tarea que lo creó.

Para llamarlo en nuestra página y que se ejecute como un WebWorker haremos:

```
var worker;

function startWorker() {
    worker = new Worker("worker_count.js");
    }
    worker.onmessage = function(event) {
      console.log = ('Recibido del worker: '+event.data);
    };
}

function stopWorker() {
  worker.terminate();
}
```

Al llamar a *startWorker* se crea el worker y cada vez que éste envíe algo se ostrará en la consola (lo que envía se recibe en *event-data*). Para finalizar un worker llamamos a su método *terminate()*.

Saber más:

- MDN - Usando WebWorkers
- w3schools - HTML5 Web Workers

6.4 Typescript

TypeScript es un lenguaje de programación libre y de código abierto desarrollado y mantenido por Microsoft. Es un superconjunto de JavaScript, que esencialmente añade tipado estático y objetos basados en clases. TypeScript extiende la sintaxis de JavaScript, por tanto cualquier código JavaScript existente debería funcionar sin problemas.

Puede que la principal diferencia entre ambos es que Typescript obliga al tipado de las variables (y por supuesto no permite cambios de tipo) lo que evita muchos errores a la hora de programar.

Algunos frameworks y librerías, como Angular, utilizan TS en lugar de JS como lenguaje, que luego es transpilado a JS a la hora de generar la aplicación para producción.

Quizá el inconveniente es que es algo más difícil que JS pero como está basado en él y la sintaxis es prácticamente igual el esfuerzo de aprender TS para un programador JS es muy pequeño.

Saber más:

- Wikipedia - Typescript
- Typescriptlang

7. jQuery

Se trata de una biblioteca que nos facilita enormemente el trabajo con el DOM ya que tiene "atajos" para muchas instrucciones, por ejemplo para poner 'Hola mundo' como contenido de un elemento cuya *id* es `mensaje`:

```
// Código con Javascript sólo
document.getElementById('mensaje').textContent = 'Hola mundo'

// Código con jQuery
$('#mensaje').text('Hola mundo')
```

Otra ventaja de jQuery es que permite trabajar con conjuntos de elementos sin tener que hacer un `forEach` (lo hace internamente). Por ejemplo para poner un escuchador que muestre un alert 'Párrafo pinchado' al hacer click sobre cualquier párrafo de la clase 'importante' tendríamos que hacer:

```
// Código con Javascript sólo
Array.from(document.querySelectorAll('p.importante'))
.forEach(parrafo => parrafo.addEventListener('click', () => alert('Párrafo pinchado))

// Código con jQuery
$('p.importante').click(() => alert('Párrafo pinchado))
```

Como vemos, básicamente nos permite hacer lo mismo pero escribiendo mucho menos código. También incluye funciones para cosas que en Javascript requieren varias líneas de código como animaciones o Ajax. Por ejemplo una pertición para mostrar en una tabla con id *posts* todos los posts del servidor *jsonplaceholder* tendremos que hacer:

```
// Código con Javascript sólo
const SERVER = 'https://jsonplaceholder.typicode.com';

function getPosts() {   // Función que pide los datos al servidor
  return new Promise(function(resolve, reject) {
    let peticion = new XMLHttpRequest();
    peticion.open('GET', SERVER + '/posts');
    peticion.send();
    peticion.addEventListener('load', function() {
      if (peticion.status === 200) {
        resolve(JSON.parse(peticion.responseText));
      } else {
        reject("Error " + this.status + " (" + this.statusText + ") en la petición");
      }
```

```
    })
    peticion.addEventListener('error', () => reject('Error en la petición HTTP'));
  })
}

function renderPosts()    // Función que los muestra en la página
      getPosts(idUser)
        .then((posts) => {
          document.querySelector('#posts tbody').innerHTML = ''; // borramos su contenido
          posts.forEach((post) => {
            const newPost = document.createElement('tr');
            newPost.innerHTML = `
                <td>${post.userId}</td>
                <td>${post.id}</td>
                <td>${post.title}</td>
                <td>${post.body}</td>`;
            document.querySelector('#posts tbody').appendChild(newPost);
          })
        })
        .catch((error) => console.error(error))
}
```

Usando jQuery es mucho más sencillo. En primer lugar no hay que hacer la función que hace la petición al servidor porque hay uns función que hace eso: `$.ajax` y sus derivadas `$.get`, `$.post`, ... Además la parte de pintar los datos es también mucho más corta:

```
// Código con jquery
const SERVER = 'https://jsonplaceholder.typicode.com';

function renderPosts()    // Función que los muestra en la página
      $.get(SERVER + '/posts')
        .done((posts) => {
          $('#posts tbody').text('');    // borramos el contenido de la tabla
          posts.forEach(post => $('#posts tbody').append(
              `<tr>
                <td>${post.userId}</td>
                <td>${post.id}</td>
                <td>${post.title}</td>
                <td>${post.body}</td>
              </tr>`
```

```
15              ))
16          })
17          .fail((error) => console.error(error))
18  }
```

Encontraréis infinidad de tutoriales por Internet donde aprender jQuery, por ejemplo unos vídeos de Didacticode que podéis encontrar en https://didacticode.com/curso/curso-de-jquery/ (tienes que registraros para tener acceso a muchos cursos de Javascript y "derivados") o directamente en su canal de Youtube.