XMLHTTPRequest

📥 **𝗖𝗹𝗮𝘀𝗲 #𝟳: 𝗫𝗠𝗟𝗛𝗧𝗧𝗣𝗥𝗲𝗾𝘂𝗲𝘀𝘁𝟳/𝟮𝟭** 📤 . 📲 **++XMLHttpRequest++** es un objeto de JS que permite hacer peticiones hacia servicios en la nube(URLs o APIs). . 📪 ++**Existen 5 estados en un llamado XMLHttpRequest:**++ .

* **0** → Se ha inicializado.
* **1** → Loading (cargando).
* **2** → Se ha cargado.
* **3** → Procesamiento si existe alguna descarga.
* **4** → Completado. .

📫 ++**Métodos y propiedades:**++ . **xmlhttp.open()** → Prepara la petición para ser enviada tomando tres parámetros: prótocolo, url, asíncrono (true). **xmlhttp.readyState** → Retorna el estado de la petición. **xmlhttp.onreadystatechange** → Un eventHandler que es llamado cuando la propiedad readyState cambia. **xmlhttp.status** → Retorna el estado de la respuesta de la petición. (200,400,500) **xmlhttp.send()** → Envía la petición. . 📬 ++**Características del protocolo http:**++ . **++Verbos++**: Los verbos indican acciones que están asociadas a peticiones y recursos, es decir, sirven para la manipulación de recursos cliente/servidor. Los Verbos http son:

* **GET** → Solicita un recurso.
* **HEAD** → Solicita un recurso pero sin retornar información, la estructura de esta petición es igual que get tanto en su headers como estatus. Es útil cuando vamos a utilizar API, para comprobar si lo que vamos a enviar esta correcto y puede ser procesado.
* **POST** → Sirve para la creación de recursos en el servidor.
* **PUT** → Actualiza por completo un recurso, reemplaza todas las representaciones actuales del recurso de destino con la carga útil de la petición.
* **PATCH** → Actualiza parcialmente un recurso.
* **DELETE** → Elimina un recurso. .

📭 ++**Los códigos de estados del servidor:**++ . El código de estado (status codes) sirve para describir el estado de la petición hecha al servidor.

* **1xx** → Indican que la petición fue recibida por el servidor, pero está siendo procesada por el servidor.
* **2xx** → Indican que la petición fue recibida, aceptada y procesada correctamente.
* **3xx** → Indican que hay que tomar acciones adicionales para completar la solicitud.
* **4xx** → Indican errores del lado del cliente que hizo mal una solicitud.
* **5xx** → Indican errores del servidor. Suelen aparecer cuando existe un fallo en la ejecución en el servidor. .

📧 **++Los códigos más comunes a la hora de interactuar con una API son:++** .

* **200** → *OK* → Indica que todo está correcto.
* **201** → *Created* → Todo está correcto cuando se hizo una solicitud POST, el recurso se creó y se guardó correctamente.
* **204** → *No Content* → Indica que la solicitud se completó correctamente pero no devolvió información. Este es común cuando se hacen peticiones con el verbo DELETE.
* **400** → *Bad Request* → Indica que algo está mal en la petición (no encontró algo).
* **401** → *Unauthorized* → Significa que antes de hacer una solicitud al servidor nos debemos autenticar.
* **403** → *Forbidden* → Indica que no tenemos acceso a ese recurso aunque se esté autenticado.
* **404** → *Not Found* → Indica que no existe el recurso que se está intentando acceder.
* **500** → *Internal Server Error* → Indica que algo falló, es un error que retorna el servidor cuando la solicitud no pudo ser procesada. . Fuente: [aquí](https://en.wikipedia.org/wiki/List_of_HTTP_status_codes) .

🖍️ **++Ejemplo en VSC:++** .

1. Ir a la consola y ubicarnos en la carpeta del proyecto y escribir el comando para instalar el paquete **++XMLHttpRequest++**: **npm i xmlhttprequest**
2. Ir al ++VSC++ y crear un archivo llamado **challenge.js** en la ruta **src/callback**. El archivo queda:

const XMLHttppRequest = requiere('xmlhttprquest'); //llamado al XmlHttpRequest const API = 'https://api.escuelajs.co/api/v1'; //API en mayúscula porque es una referencia que no va a cambiar function fetchData(urlApi, callback){ //urlApi: no confundir y colocar API let xhttp = new XMLHttppRequest(); //referencia a new XMLHttpRequest xhttp.open('GET', urlApi, true); //petición "obtener" con true para habilitarlo xhttp.onreadystatechange = function(event) { //escucha diferentes estados de la solicitud y conocer cuando está disponible la información if(xhttp.readyState === 4) { //si el estado ha sido completada la llamada if(xhttp.status === 200 ){ //el servido responde de forma correcta callback(null, JSON.parse(xhttp.responseText)); //dentro de xhttp.responseTex recibimos lo que entrega el servidor en texto y se hace la transformación en JSON } } else { const error = new Error('Error' + urlApi); return callback(error,null); //es null porque no se está regresando ningún dato } } xhttp.send(); }

const XMLHttppRequest = requiere('xmlhttprquest'); //llamado al XmlHttpRequest

const API = 'https://api.escuelajs.co/api/v1'; //API en mayúscula porque es una referencia que no va a cambiar

function fetchData(urlApi, callback){ //urlApi: no confundir y colocar API

let xhttp = new XMLHttppRequest(); //referencia a new XMLHttpRequest

xhttp.open('GET', urlApi, true); //petición "obtener" con true para habilitarlo

xhttp.onreadystatechange = function(event) { //escucha diferentes estados de la solicitud y conocer cuando está disponible la información

if(xhttp.readyState === 4) { //si el estado ha sido completada la llamada

if(xhttp.status === 200 ){ //el servido responde de forma correcta

callback(null, JSON.parse(xhttp.responseText)); //dentro de xhttp.responseTex recibimos lo que entrega el servidor en texto y se hace la transformación en JSON

}

} else {

const error = new Error('Error' + urlApi);

return callback(error,null); //es null porque no se está regresando ningún dato

}

}

xhttp.send();

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

const XMLHttpRequest = require("xmlhttprequest").XMLHttpRequest;

const API = "https://api.escuelajs.co/api/v1";

//funcion principal que obtendrá la informacion del producto como un objeto

function fetchData(urlApi, callback) {

//inicializar un objeto de tipo XMLHttpRequest

let xhttp = new XMLHttpRequest();

//El metodo .open realiza la petición de apertura de comunicación, el metodo puede ser 'GET' o 'POST', luego se envia la URL, si es asincrono (true o false), usuario y contraseña. En esta caso solo se utiliza el metodo, la url y async

xhttp.open('GET', urlApi, true);

//en este metodo Almacena el nombre de la función que se ejecutará cuando el objeto XMLHttpRequest cambie de estado

xhttp.onreadystatechange = function (event) {

//el atributo readyState define el estado del objeto XMLHttpRequest

//0 No inicializado

//1 Loading

//2 ejecutado

//3 interactuando

//4 completado

if (xhttp.readyState === 4) {

//si la respuesta de la API es exitosa (200 Ok)

if (xhttp.status === 200) {

//se ejecuta el callback recibiendo como argumentos un objeto, como la respuesta de la API es un texto plano, el metodo JSON.parse tranformará este texto en un objeto.

//El atributo devuelve un DOMString que contiene la respuesta a la consulta como un texto o null si la consulta no tuvo exito o aun no ha sido completada.

callback(null, JSON.parse(xhttp.responseText));

//si la respuesta de la API no es exitosa se captura el error

} else {

//se inicializa un objeto de tipo Error donde se le envian como argumentos un mensaje de error y la URL de la API para conocer en dónde se produjo el error

const error = new Error("Error" + urlApi);

//se ejecuta el callback recibiendo como argumentos el error y null debido a que no se pudo obtener el objeto

return callback(error, null);

}

}

//el método .send() envia la petición al servidor

}

xhttp.send();

}

//se invoca el metodo fetchData() pasandole como argumentos la varible API concatenada con la cadena 'products' para acceder a la URL de la API deseada, y una función anónima que recibe 2 parámetros (un objeto de error y un arreglo que almacena todos los objetos traidos por la API).

fetchData(`${API}/products`, function (error1, data1) {

//se valida si existe un error, en caso de que exista se detiene el proceso y se imprime el error

if (error1) return console.error(error1);

//se invoca nuevamente la función fetchData con el fin de acceder a un objeto puntual del arreglo data1, se envia como parámetros la url de la API apuntando al atributo del primer objeto de arreglo data1 y nuevamente una función anónima.

fetchData(`${API}/products/${data1[0].id}`, function (error2, data2) {

//si en este punto se identifica un error se imprime en consola y se detiene el proceso

if (error2) return console.error(error2);

//Se invoca nuevamente la funcion fetchData con el fin de acceder a la categoria, se envían como parametros la url de la API con la concatenación de 'Categories' y el atributo Id de categoria del objeto data2 de la función anterior

//en este caso puntual se hace uso de Optional Caining el cual hace una evalucación de las propiedades de un objeto y en vez de arrojar un error devuelve undefined en caso que la propiedad no exista o sea null.

//igual que las anteriores e envia una funcion anonima con 2 argumentos, un objeto Error y un objeto de datos

fetchData(`${API}/categories/${data2?.category?.id}`, function (error3, data3) {

//se valida si existe error, en caso de que exista se detiene el proceso y se imprime el error

if (error3) return console.error(error3);

//Se imprime el objeto en la posición 1 del arreglo de los objetos obtenidos en el metodo invocado inicialmente

console.log(data1[0]);

//Se imprime el titulo del objeto que se consultó en la seguna invocación de la función

console.log(data2.title);

//Se imprime el nombre de la categoria a la que pertenece el objeto que se consultó en la seguna invocación del método.

console.log(data3.name);

});

});

});