# Manual técnico ajax

Este documento busca aclarar de la manera más breve posible el uso de Ajax en JavaScript. Además de ejemplificar el uso de Ajax en conjunto con SweetAlert2, incluyendo igualmente una explicación de esta librería.

Jquery es una biblioteca de JavaScript que facilita el desarrollo de aplicaciones web. Proporciona un conjunto de herramientas para trabajar con HTML, CSS y JavaScript. Entre las funciones que proporciona se encuentran las funciones AJAX, que permiten a los desarrolladores realizar peticiones HTTP **asíncronas** (consultar definición) desde el lado del cliente. jQuery ofrece una interfaz sencilla para trabajar con AJAX, lo que facilita el proceso de crear aplicaciones web interactivas y dinámicas. Esto permite que solo se actualicen los datos necesarios, lo que mejora la velocidad y la experiencia del usuario. Sin embargo, no es necesario utilizar jQuery para trabajar con AJAX, ya que se pueden hacer peticiones AJAX usando solo JavaScript.

Sin embargo, jQuery ofrece una interfaz simplificada para trabajar con AJAX, lo que puede hacer el proceso de crear solicitudes y manejar respuestas más fácil y legible. Algunas de las ventajas de utilizar jQuery para AJAX son: compatibilidad con navegadores antiguos, manejo de eventos y **callbacks** (consultar segundo apartado), y una sintaxis sencilla para el manejo de peticiones y respuestas.

Ahora bien, para dejar más claro esto, no hay otra cosa mejor que los ejemplos para mostrar el uso de ambas librerías.

**AJAX.**

AJAX como tal no es una librería, AJAX es un método incorporado en JavaScript para hacer peticiones y manejar los datos en respuesta a dichas peticiones. Una petición AJAX se puede realizar utilizando solo JavaScript utilizando la interfaz *XMLHttpRequest* o la nueva *Fetch API*. Para hacer una petición GET con *XMLHttpRequest*, el código podría ser algo así:

var xhr = new XMLHttpRequest();

xhr.open("GET", "example.com", true);

xhr.onreadystatechange = function() {

    if (xhr.readyState === 4 && xhr.status === 200) {

        var data = xhr.responseText;

        // procesar la respuesta

    }

};

xhr.send();

Claro, esto es sólo un ejemplo. Ya que el manejo de peticiones con Jquery tiene otra estructura. Una petición AJAX a través de Jquery luce algo así:

$.ajax({

    url: 'https://example.com/api/data',

    type: 'GET',

    success: (data) => {

      console.log(data);

    }

});

En este ejemplo reducido, estamos utilizando el método **$.ajax()** de jQuery para enviar una solicitud HTTP GET a la URL '<https://example.com/api/data>'. Si la solicitud es exitosa, la función especificada en el campo **success** (POO) se ejecutará y se imprimirá la respuesta en la consola. La función **$.ajax()** toma un objeto de opciones como argumento. En este caso, estamos especificando la URL de la solicitud, el tipo de solicitud (**GET**) y la función a ejecutar en caso de éxito. Existen varias opciones disponibles (*enseguida se explican con ejemplos*), como **data** que sirve para enviar datos en la petición, **error** para manejar errores, **headers**, para enviar headers personalizados, entre otras.

Es importante mencionar que, al hacer peticiones a otro dominio, debes tener en cuenta la *política de CORS* (acceso cruzado a recursos) que restringe el acceso a ciertos recursos en otro dominio (por seguridad y prevención de *XSS*).

Un ejemplo más detallado podría ser el siguiente:

$.ajax({

    url: 'https://example.com/api/data',

    data: {

        param1: value1,

        param2: value2

    },

    type: 'POST',

    success: (data) => {

      console.log(data);

    },

    error: (jqXHR, textStatus) => {

        // Hacer algo para el manejo del error...

    }

});

Donde se hace una petición a través del método **$.ajax()**, de tipo **POST**, especificando en este caso la **data** y la función de **error**. El objeto data contiene los datos que se enviaran a la dirección '<https://example.com/api/data>', indicada en el parámetro **url**. **data** es una opción que se puede pasar al método **$.ajax()** de jQuery para enviar datos en la petición HTTP. Los datos se pueden enviar en varios formatos diferentes, como *JSON*, *texto plano*, o como una *cadena de consulta*.

En el ejemplo anterior, los datos que se están enviando son un *objeto JavaScript* con dos pares clave-valor: **{param1: value1, param2: value2}**. Estos datos se enviarán en el cuerpo de la solicitud HTTP **POST**, y pueden ser recibidos y procesados ​​por el servidor. Dependiendo del tipo de petición, el formato de los datos y la configuración del servidor, puedes enviar los datos de diferentes formas. Por ejemplo, si estás haciendo una petición **GET**, los datos se envían en la URL como parámetros de consulta. Es importante mencionar que, para algunos tipos de peticiones, como GET, no es necesario enviar **data** ya que esta petición solo solicita información y no envía ningún tipo de dato.

En el caso de la función **error**, hay dos parámetros que se reciben para el manejo de errores:

* **jqXHR**
* **textStatus**

**jqXHR** por su parte, es un objeto que representa la respuesta HTTP devuelta por el servidor. Es una subclase de la clase **XMLHttpRequest** y proporciona algunas características adicionales proporcionadas por jQuery. Puedes acceder a las propiedades de la respuesta HTTP, como el código de estado, los encabezados, la respuesta, etc.

Para el caso de **textStatus**, se puede decir que este es un string, o una cadena que indica el estado de la respuesta. Puede ser "**success**", "**notmodified**", "**error**", "**timeout**", "**abort**" o "**parsererror**", dependiendo de la situación. Este valor te permite determinar qué tipo de error ocurrió y puede ser útil para manejar errores específicos. Por ejemplo, si el servidor devuelve un código de estado *HTTP 500* (error interno del servidor), puedes mostrar un mensaje de error específico al usuario. Si el error es "**timeout**", puedes mostrar un mensaje diferente al usuario indicando que la solicitud ha tardado demasiado tiempo.

En resumen, **jqXHR** y **textStatus** son dos objetos que proporcionan información adicional sobre la respuesta HTTP devuelta por el servidor, que pueden ser utilizados para manejar errores de manera más precisa.

Con todo esto es que Jquery facilita el uso de peticiones AJAX, además de extender su compatibilidad con navegadores más antiguos, usando el mismo método **$.ajax()**.

**Asincronía y callbacks (JavaScript).**

En JavaScript la asincronía es una característica fundamental que permite a los programas continuar ejecutando otras tareas mientras se espera a que una operación dada termine.

En JavaScript, las operaciones que pueden tardar un tiempo en completarse, como las peticiones de red o el acceso al disco, se realizan de forma asíncrona. En lugar de bloquear el programa y esperar a que la operación termine antes de continuar, el programa puede continuar ejecutando otras instrucciones mientras se espera a que la operación asíncrona se complete. Una forma de manejar tareas asíncronas es utilizando "**callbacks**". Un **callback** es una función que se pasa como argumento a otra función y se ejecuta una vez que la función principal ha terminado. Es decir, es una función que se utiliza como un mecanismo para que una función "A" informe a otra función "B" que ha finalizado su ejecución.

Por ejemplo, imagine que tiene una función que realiza una operación costosa, como una petición de red o el acceso al disco. En lugar de bloquear el programa y esperar a que la operación termine antes de continuar, puede pasar una función **callback** como argumento para que sea ejecutada una vez que la operación costosa haya finalizado. Aquí un ejemplo:

function operacionCostosa(callback) {

    // Realizar operación costosa

    // ...

    // Una vez finalizada, ejecutar el callback

    callback();

}

// Función en respuesta (callback)

operacionCostosa(function() {

    console.log("La operación costosa ha finalizado");

});

Otra forma de manejar tareas asíncronas es mediante "**promesas**" y "**async/await**" son una forma más moderna de manejar tareas asíncronas, que proporciona una sintaxis más clara y fácil de entender. Una **promesa** (**promise**) es un objeto que representa un valor que aún no está disponible, pero que se espera que esté disponible en algún momento en el futuro. Una **promesa** tiene tres estados posibles: *pendiente*, *cumplida* y *rechazada*. Una **promesa** se crea utilizando la función **Promise** ***constructor***, y se le pasan dos **callbacks**, ***resolve*** y ***reject***. El **callback** ***resolve*** se ejecuta cuando la promesa se cumple, y el **callback *reject*** se ejecuta cuando la promesa es rechazada. Aquí tienes un ejemplo de una promesa que se cumple después de un segundo:

let promise = new Promise((resolve, reject) => {

    // Se simula un tiempo de espera

    setTimeout(() => {

        resolve("La promesa se ha cumplido");

    }, 1000);

});

promise.then((result) => {

    console.log(result);

});

En este ejemplo, se está creando una **promesa** y se está especificando que se cumplirá después de 1 segundo. La promesa se está asignando a la variable **promise** y se está ejecutando el *metodo* ***then***, el cual recibe como argumento un **callback**, que se ejecutará una vez que la **promesa** se haya cumplido. Ahora bien, **async/await** es una forma de manejar promesas de forma más fácil y legible. **async** es una palabra clave que se utiliza antes de una función para indicar que esta función puede contener código asíncrono. **await** se utiliza para esperar a que una promesa se cumpla antes de continuar ejecutando el código. Para dejar más clara la diferencia, aquí hay un ejemplo de cómo se podría escribir el ejemplo anterior utilizando **async/await**:

async function example() {

    let result = await new Promise((resolve) => {

        setTimeout(() => {

            resolve("La promesa se ha cumplido");

        }, 1000);

    });

    console.log(result);

}

example();

En este ejemplo, la función **example()** está marcada como **async**. Dentro de esta función, se está utilizando la palabra clave **await** para esperar a que una **promesa** se cumpla antes de continuar ejecutando el código. En resumen, las promesas son objetos que representan un valor que aún no está disponible, pero que se espera que esté disponible en algún momento en el futuro, y **async/await** es una forma moderna y legible de manejar **promesas**.

En conclusión, la **asincronía** en JavaScript permite a los programas continuar ejecutando otras tareas mientras se espera a que una operación dada termine. Esto permite una mejor eficiencia y un mejor uso de los recursos del sistema.

**Sweet Alert 2.**

Ahora bien, ya explicado lo que hace AJAX, un ejemplo practico de AJAX (con jquery) es cuando se mandan peticiones a través de una función **callback** de **SweetAlert2**. Pero antes que nada, hay que aclarar primero que es **SweetAlert2**. Ésta, es una biblioteca de JavaScript que proporciona una interfaz de alerta mejorada y más personalizable que la interfaz de alerta predeterminada del navegador. Es una biblioteca de alertas de JavaScript *modal* que proporciona una interfaz de usuario más atractiva y personalizable que la interfaz de alerta predeterminada del navegador. Con **SweetAlert2**, puedes crear alertas, confirmaciones y diálogos modales con varios tipos de contenido, como texto, imágenes y elementos HTML. Además, se pueden personalizar la apariencia de los diálogos modales mediante opciones de configuración. También proporciona una serie de funciones para interactuar con el usuario, como la posibilidad de mostrar una carga en un *modal de diálogo*, o mostrar un botón de carga. Para dejar esto aún más claro, aquí un ejemplo del uso de **SweetAlert2**:

Swal.fire({

    title: 'Custom Title',

    text: 'Custom Text',

    icon: 'success',

    showCancelButton: true,

    confirmButtonText: 'OK',

    cancelButtonText: 'Cancel',

    footer: '<a href>Why do I have this issue?</a>'

});

En este ejemplo se está creando un ***objeto*** con varias opciones de configuración. Este *objeto* se le está pasando como argumento al método **fire()** de **SweetAlert2**. Algunas de las opciones disponibles son:

* **title:** título que se mostrara en el dialogo modal.
* **text:** texto que se mostrara en el dialogo modal.
* **icon:** icono que se mostrara en el dialogo modal.
* **showCancelButton:** boolean para mostrar o no un boton de cancelar.
* **confirmButtonText:** texto que se mostrara en el boton de confirmar.
* **cancelButtonText:** texto que se mostrara en el boton de cancelar.
* **footer:** pie de página que se mostrara en el dialogo modal.
* **showCloseButton**: permite mostrar o no un botón de cierre en la esquina superior derecha del diálogo modal. Por defecto, esta opción está desactivada.
* **position**: permite especificar la posición en la que se mostrará el diálogo modal. Las opciones disponibles son: "top", "top-start", "top-end", "center", "center-start", "center-end", "bottom", "bottom-start" y "bottom-end". Por defecto, la posición es "center".
* **animation**: permite especificar la animación que se utilizará al mostrar el diálogo modal. Las opciones disponibles son: "pop", "slide-from-top", "slide-from-bottom", "slide-from-top", "slide-from-bottom". Por defecto, la animación es "pop".
* **timer**: permite especificar el tiempo en milisegundos después del cual el diálogo modal se cerrará automáticamente. Por defecto, este valor es 0, lo que significa que el diálogo.

Hay una opción que suele usarse con más frecuencia, es la de **icon**. **icon** permite especificar el icono que se mostrará en el diálogo modal. Las opciones disponibles son:

* '**success**': Muestra un icono de éxito (una marca de verificación)
* '**error**': Muestra un icono de error (un signo de exclamación dentro de un círculo rojo)
* '**warning**': Muestra un icono de advertencia (un signo de exclamación dentro de un triángulo amarillo)
* '**info**': Muestra un icono de información (un signo de exclamación dentro de un círculo azul)
* '**question**': Muestra un icono de pregunta (un signo de interrogación dentro de un círculo gris)

También puedes utilizar una URL para mostrar una imagen personalizada como icono. La opción **html** permite mostrar contenido HTML en el diálogo modal. Esto te permite mostrar contenido personalizado, como formularios, imágenes, etc. Además de haber muchas otras opciones disponibles, como *personalizar* *el color de los botones*, mostrar un *spinner de carga*, etc.

Es importante mencionar que la biblioteca **SweetAlert2** se basa en **promesas**, por lo que se pueden usar métodos como **then()** y **catch()** para interactuar con el diálogo modal una vez que se ha mostrado.

En resumen, **SweetAlert2** es una biblioteca de alertas modales personalizadas y atractivas para JavaScript que proporciona una interfaz de usuario más atractiva y personalizable que la interfaz de alerta predeterminada del navegador. Es una biblioteca fácil de usar y personalizar, con una gran cantidad de opciones y una gran cantidad de funciones para interactuar con el usuario.