

*Librerías*

**¿Qué es una librería?**

Una librería en Javascript es un código pre-escrito que facilita el desarrollo de aplicaciones. Podemos pensar las librerías como ***pequeños programas*** escritos por terceros que podemos incorporar a nuestra aplicación para resolver problemas determinados.

Dado al uso cada vez mayor y a la escalabilidad de Javascript en el desarrollo de aplicaciones, se vuelve necesario agilizar los desarrollos implementando interfaces dinámicas. Aquí es donde se desarrollan librerías que funcionan prácticamente como cajas de herramientas que resuelven problemas recurrentes de forma rápida y eficiente, y podemos incorporarlas y utilizarlas a discreción según nuestra demanda.

Hay muchas librerías de JS disponibles que resuelven problemas de diverso tipo, como validaciones de datos, interfaces visuales, manejos de fechas, AJAX, y demás. Algunas librerías exceden el uso de JS e implementan también HTML y CSS en su desarrollo.

**Implementación**

Las librerías son códigos que podemos incorporar a nuestro proyecto como archivos. Por lo tanto, se vinculan a nuestra aplicación en el HTML como cualquier otro script de Javascript, a través de un CDN o habiendo descargado los archivos de la librería, según corresponda:

| <script src="js/libreria.js"></script> |
| --- |

Con un CDN:

| <<script src="https://cdnjs.cloudflare.com/ajax/libs/moment.js/2.29.1/moment.min.js">  </script> |
| --- |

**Archivos minificados**

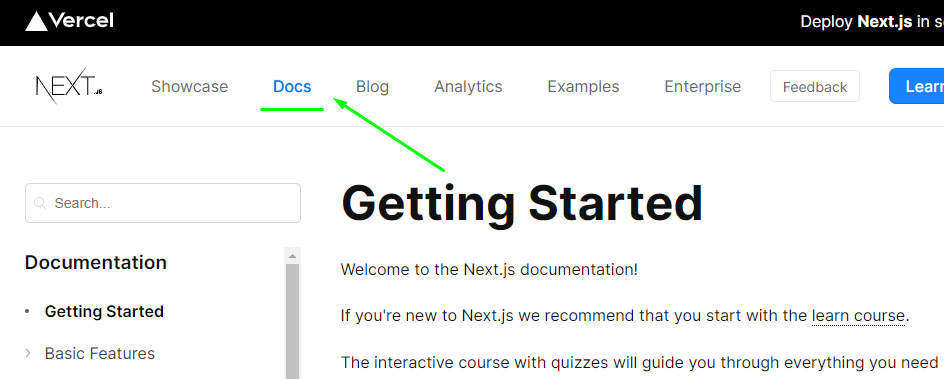
Al cargar librerías externas, hay una práctica muy común que consiste en cargarlas en formato minificado, o minified. Eso significa, que el contenido del archivo JS se encuentra escrito todo en una única línea y de corrido, sin saltos de línea ni espacios innecesarios. Esto se usa para achicar el peso del archivo, agilizando su tiempo de carga; y para evitar la edición del archivo JS ya que se vuelve casi ilegible en este formato, aunque es reversible.

Por lo general, todas las librerías suelen ofrecer los dos archivos para usar: el archivo .js común, y el minificado.

**Documentación**

Existen una gran cantidad de librerías disponibles que podemos incorporar a nuestros proyectos, y cómo desarrolladores, más que dominar alguna librería en particular, tenemos que tener la capacidad de saber cómo incorporar cualquiera de ellas. Para esto hemos de acostumbrarnos a trabajar con la **documentación** de las librerías, que vendrían a ser un *manual de instrucciones* sobre cómo implementarlas y manejarlas.

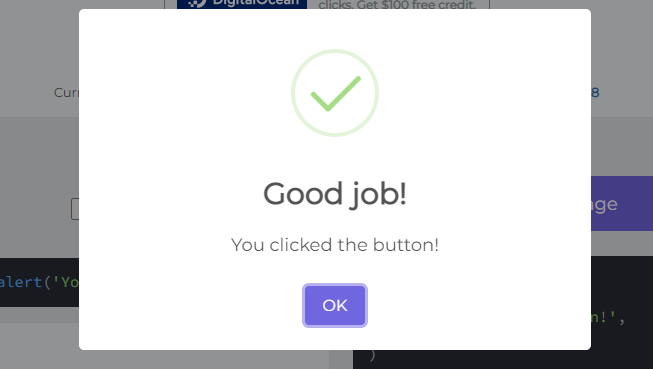
Cualquier librería que valga la pena incorporar a nuestro proyecto viene mínimamente documentada. Esto es que incluye un apartado donde explica y define qué es esta librería, cómo funciona, cómo manejarla e integrarla a nuestro trabajo, ejemplos de uso, etc.



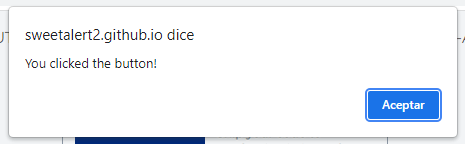
Este tipo de enlaces suele estar siempre a la vista y disponible en cualquier librería. Debemos habituarnos a la investigación y uso de este tipo de herramientas. No hay que dejarse intimidar por el tamaño que pueden llegar a tener estos documentos y habituarse a su lectura, ya que un buen desarrollador se destaca por poseer esta habilidad.

*Sweet Alert*

Pasemos a la implementación de nuestra primer librería, Sweet Alert ([SweetAlert2](https://sweetalert2.github.io/)) . Esta nos permite crear *alertas* personalizadas muy atractivas y sencillas, customizables e interactivas.



Viene en reemplazo del típico *alert()* que usábamos en el navegador. La alerta es una herramienta útil para comunicarse con el usuario, pero la tradicional es nada estética y no ofrece mayor utilidad que el despliegue del mensaje:



Para esto llega Sweet Alert a salvarnos con una opción de fácil implementación, atractiva e interactiva.

**Instalación**

Repasando los pasos para implementar una librería, lo primero es integrarla a nuestro trabajo. Recordemos que son programas escritos en archivos externos que podemos utilizar a discreción, pero deben estar vinculados primero.

Lo hacemos agregando el script correspondiente a nuestro html, con el CDN que nos indique la documentación o bien descargándolo. Siguiendo la documentación, en el item *installation*  tenemos las instrucciones de cómo proceder:

| <script src="//cdn.jsdelivr.net/npm/sweetalert2@11"></script> |
| --- |

Una vez agregado, ya estamos listos para utilizarla.

**Uso**

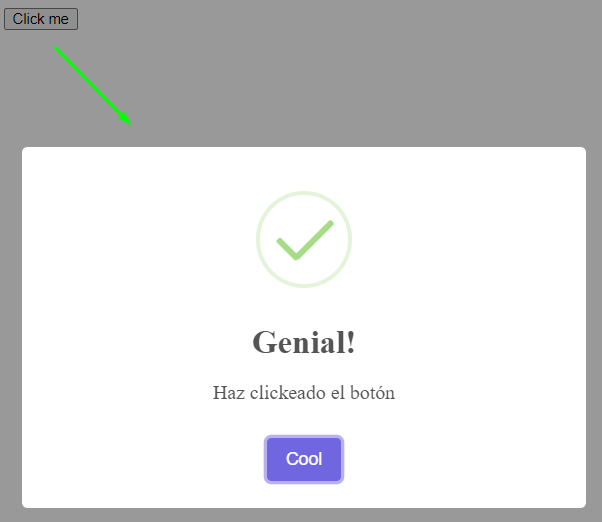
Una vez incorporado, podemos *disparar* estos alertas a través del método *.fire* del objeto global *Swal:*

| Swal.fire({  title: 'Error!',  text: 'Do you want to continue?',  icon: 'error',  confirmButtonText: 'Cool'  }) |
| --- |

La alerta se genera y dispara en el momento. Nótese que el método *fire* recibe un *objeto* por parámetro. Esta es la configuración con la cual customizamos la alerta. Puede recibir distintas propiedades y valores generando distintos resultados.

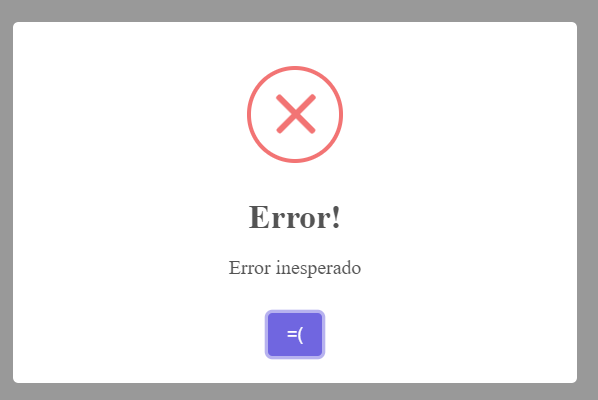
Por lo general, vamos a querer las alertas vinculadas a determinados eventos. Por ejemplo, el click de un botón:

| <body>    <button id="myBtn">Click me</button>  <script src="//cdn.jsdelivr.net/npm/sweetalert2@11"></script>  <script src="app.js"></script>  </body>  // JS  const btn = document.querySelector('#myBtn')  btn.addEventListener('click', () => {  Swal.fire({  title: 'Genial!',  text: 'Haz clickeado el botón!',  icon: 'success',  confirmButtonText: 'Cool'  })  }) |
| --- |

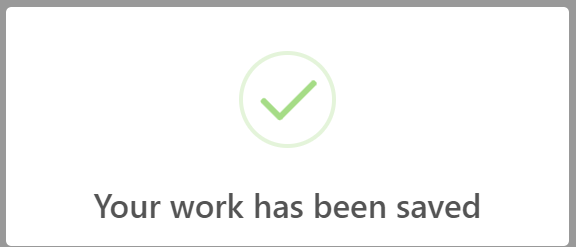


***nota:*** *sweet alert viene estilado con bootstrap, por lo que es recomendable tenerlo integrado para mejor resultado visual.*

Si analizamos el alert generado, tiene un ícono (success), un título, un texto, y un botón de confirmación. Si modificamos los valores de estas propiedades, podemos generar un alert con otras características:



| btn.addEventListener('click', () => {  Swal.fire({  title: 'Error!',  text: 'Error inesperado',  icon: 'error',  confirmButtonText: ' =( '  })  }) |
| --- |

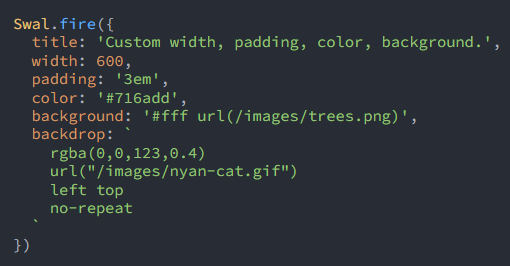
El alert tiene una estructura por default. En ella se encuentra el botón de confirmación y la posición centrada. Podemos agregar a la configuración la propiedad *shoConfirmButton: false,* para ocultar el botón, modificar la *position*, y agregar un *timer* con valor en milisegundos para que el alert se cierre automáticamente:

| Swal.fire({  position: 'top-end',  icon: 'success',  title: 'Your work has been saved',  showConfirmButton: false,  timer: 1500  }) |
| --- |

Veremos el alert ubicado en la parte superior derecha del viewport. En la documentación se listan todas las propiedades y valores posibles que se pueden definir para configurar el alert.

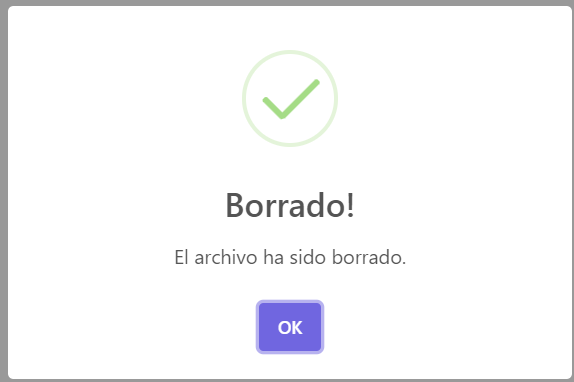
Se pueden agregar imágenes a las alertas a través de la propiedad *imageUrl*, y también asignar el *width, height* y *alt* de la misma:

| Swal.fire({  title: 'Sweet!',  text: 'Modal with a custom image.',  imageUrl: 'https://unsplash.it/400/200',  imageWidth: 400,  imageHeight: 200,  imageAlt: 'Custom image',  }) |
| --- |

O incluso modificar el background del alert y el fondo, ancho, padding, y demás. Sólo es un ejemplo de lo fácil y versátil que puede ser el funcionamiento de Sweet Alert y cómo podemos crear los modales a discreción:

En la sección de *Recipe Gallery* de la documentación hay muchos ejemplos de todo lo que se puede realizar. A veces basta con copiar uno y modificar algunas propiedades para obtener lo que queremos. Recordemos que es importante definir el *cuando* queremos disparar los alert, definiendo correctamente los eventos y el comportamiento esperado en nuestro script.

Podemos también generar una interacción más completa al mostrar botones de confirmación y cancelación, y controlar qué se hace luego en cada caso. La documentación define claramente cómo debe hacerse esto, ya que trabaja con cierto estado condicional definido por la librería que debemos controlar.



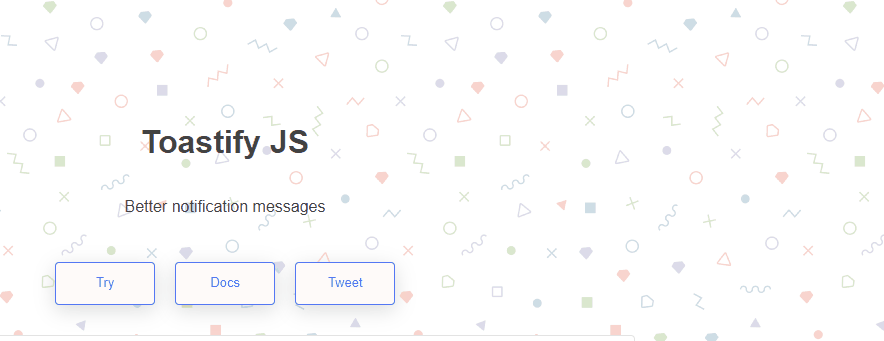
| btn.addEventListener('click', () => {    Swal.fire({  title: 'Está seguro de eliminar el producto?',  icon: 'warning',  showCancelButton: true,  confirmButtonText: 'Sí, seguro',  cancelButtonText: 'No, no quiero'  }).then((result) => {  if (result.isConfirmed) {  Swal.fire({  title: 'Borrado!',  icon: 'success',  text: 'El archivo ha sido borrado'  })  }  })  }) |
| --- |

Luego del primer *.fire()* podemos concatenar un método *.then()* siguiendo la sintaxis del ejemplo, que recibe por parámetro el *result* del modal anterior. Evaluamos el estado de ese resultado y con eso podemos definir qué hacer en el caso de que se haya apretado el botón de confirmación o de cancelación. En este caso, si el estado es de *isConfirmed* disparamos un nuevo modal con un mensaje, de lo contrario no hacemos nada aunque podríamos definirlo también.

¿Qué es este ***.then() ?*** Tiene que ver con algo que llamamos **promesas** y lo veremos en detalle la clase que viene!

*Toastify*

Toastify es una librería que nos permite mostrar mensajes de notificación personalizados.



La configuración básica indica hacer un llamado a Toastify() con un objeto de configuración, y al método *.showToast()* concatenado para dispararlo, y asignando la propiedad *text* con el mensaje a mostrar, y *duration* en milisegundos para el fadeout.

| btn.addEventListener('click', () => {    Toastify({  text: "Probando toast!",  duration: 3000  }).showToast();  }) |
| --- |

La documentación de ésta librería es más sencilla que la de Sweet Alert, escrita en el README.md de su repositorio: (<https://github.com/apvarun/toastify-js/blob/master/README.md> )

El método de llamado es siempre el mismo, pero según la cantidad de propiedades y valores que definamos el resultado será diferente.

Por ejemplo, podemos empezar modificando la posición del Toast a través de las propiedades *gravity (‘top’* o ‘*bottom’*), y *position (‘left’* o *‘right’)*:

| btn.addEventListener('click', () => {    Toastify({  text: "Probando toast!",  duration: 3000,  gravity: 'bottom',  position: 'left'  }).showToast();  }) |
| --- |

Podemos modificar el background de nuestro Toast a través de la propiedad *style* del objeto de configuración:

| Toastify({  text: "Probando toast!",  duration: 3000,  gravity: 'bottom',  position: 'left',  style: {  background: 'linear-gradient(to right, #00b09b, #96c92d)'  }  }).showToast(); |
| --- |

O bien, podemos estilarlo a nuestro gusto definiendo *clases* para la notificación, que asignamos a través de la propiedad *className:*

| Toastify({  text: "Probando toast!",  duration: 3000,  gravity: 'bottom',  position: 'left',  className: 'notificacion my-toast'  }).showToast(); |
| --- |

A través de la propiedad *onClick* podemos definir una función de callback que se ejecuta al clickear el toast, generando interacciones interesantes:

| btn.addEventListener('click', () => {    Toastify({  text: "Probando toast!",  duration: 3000,  onClick: () => {    Toastify({  text: 'Clickeaste un Toast!',  duration: 1500,  position: 'left'  }).showToast()  }  }).showToast();  }) |
| --- |

Con la propiedad *destination* podemos definir URLs de destino al clickear el Toast. Esto es útil para poder direccionar el usuario a páginas de interés acordes al mensaje mostrado:

| btn.addEventListener('click', () => {    Toastify({  text: "Click aquí para ir a Coder!",  duration: 3000,  destination: 'https://www.coderhouse.com'  }).showToast();  }) |
| --- |

*LUXON*

Luxon ([Luxon 2.x](https://moment.github.io/luxon)) es una librería que permite trabajar con fechas y horas de una forma sencilla. Con Luxon puedes formatear, parsear, sumar y restar fechas, entre otras funciones.

**Instalación**

Puedes agregarlo a tu proyecto siguiendo las instrucciones de la documentación, descargando el script correspondiente y vinculándolo al html.



O bien utilizando el CDN:

| <script src="https://cdn.jsdelivr.net/npm/luxon@2.3.0/build/global/luxon.min.js"></script> |
| --- |

Esto nos provee la variable global **luxon** de la cual podemos acceder a todos los métodos necesarios.

**Usos**

La clase principal en luxon es DateTime. Para ser más eficientes/ordenados, podemos referenciarla en una variable global para facilitar su acceso:

| const DateTime = luxon.DateTime |
| --- |

Un *DateTime* representa un milisegundo específico en el tiempo, junto con una zona horaria. Podemos crear nuestro primer DateTime con el método *.local()*, que recibe argumentos desde el año hasta los milisegundos. En principio es similar al objeto Date de JS, pero trae diversos métodos útiles:

| const dt = DateTime.local(2022, 1, 25, 12, 10)  // 25 / 01 / 2022 - 12:10hs |
| --- |

Podemos crear un objeto de fecha con la fecha actual a través de *DateTime.now()*, que es equivalente a llamar a *DateTime.local()* sin argumentos.

También podemos crear fechas a través de un objeto, pasando las distintas características por propiedades y valores:

| const dt = DateTime.fromObject(  { day: 22, hour: 12, month: 2 },  { zone: 'America/Buenos\_Aires', numberingSystem: 'beng' }  )  console.log( dt.toString() )  // 2022-02-22T12:00:00.000-03:00 |
| --- |

Vemos que el método *fromObject* recibe dos objetos por parámetro. El primero donde definimos numéricamente la fecha y hora, y el segundo donde aclaramos la zona horaria y el tipo de numeración.

También podemos crear fechas a partir de un string siguiendo la norma ISO, con el método *fromISO:*

| DateTime.fromISO("2017-05-15") //=> May 15, 2017 at midnight  DateTime.fromISO("2017-05-15T08:30:00") //=> May 15, 2017 at 8:30 |
| --- |

**Accediendo a las fechas**

Si hicimos *console.log* de los ejemplos anteriores notamos que no obtenemos una presentación clara, útil, de las fechas creadas. Para poder recuperar los valores en un formato legible debemos aplicar algunos métodos específicos para esto. Algo que sí queremos es recuperar la fecha como string, y para eso aplicamos el método *.toString()* al objeto creado:

| const now = DateTime.now()  console.log( now.toString() )  // 2022-01-25T12:31:00.775-03:00 |
| --- |

Podemos acceder a los distintos componentes de la fecha creada a través de propiedades específicas:

| const dt = DateTime.now();  dt.year //=> 2022  dt.month //=> 1  dt.day //=> 25  dt.second //=> 22  dt.weekday //=> 2 |
| --- |

Y hay otros accesos posibles y útiles. La totalidad de ellos se puede revisar en la documentación de Luxon para decidir cuál es el que necesitas:

| const dt = DateTime.now();  console.log( dt.zoneName ) // America/Buenos\_Aires  console.log( dt.daysInMonth ) // 31 |
| --- |

**Formatear la fecha**

En general vamos a querer mostrar nuestra fecha en un formato legible acorde a nuestra necesidad. Para esto, el más común es el método *.toLocaleString()* que ajusta la fecha a un formato sencillo y genérico. Sin embargo, Luxon posee una lista de *presets* con distintos formatos posibles, el cual podemos elegir y enviar por parámetro al método *.toLocaleString()*. Adicionalmente, puedes concatenar previamente el método *.setLocale* para cambiar el idioma de presentación.



| const dt = DateTime.now();  dt.toLocaleString() // 25/1/2022  dt.toLocaleString(DateTime.DATE\_FULL) // 25 de enero de 2022  dt.toLocaleString(DateTime.TIME\_SIMPLE) // 12:51  dt.setLocale('en').toLocaleString(DateTime.DATE\_FULL)  // January 25, 2022 |
| --- |

Puedes acceder al listado de *presets*  aquí <https://moment.github.io/luxon/#/formatting?id=presets>

**Transformación**

Los objetos de Luxon son inmutables. Esto significa que los métodos de transformación no modifican la variable sobre la cual los aplicamos, sino que retornan nuevas instancias alteradas de ellos.

Con los métodos *plus* y *minus* podemos obtener cálculos sobre las fechas de forma sencilla. Pasamos un objeto con la cantidad de días, meses, horas, minutos, etc., que queramos sumar o restar a la fecha de referencia:

| const dt = DateTime.now();  console.log( dt.toLocaleString(DateTime.DATETIME\_SHORT) )  // 25/1/2022 14:21  const suma = dt.plus({ hours: 5, minutes: 15 })  console.log(suma.toLocaleString(DateTime.DATETIME\_SHORT))  // 25/1/2022 19:36  const resta = dt.minus({ month: 2, days: 10 })  console.log(resta.toLocaleString(DateTime.DATETIME\_SHORT))  // 15/11/2021 14:21 |
| --- |

Se pueden crear nuevas instancias reemplazando propiedades específicas:

| const dt = DateTime.now()  dt.set({hour: 3}).hour //=> 3 |
| --- |

**La clase Duration**

La clase Duration representa una cantidad de tiempo, como puede ser “3 horas y 15 minutos”. Se crea a través del método *.fromObject* enviando un objeto con las características deseadas. Se puede sumar o restar a un objeto DateTime, y también tiene sus propios getters:

| const Duration = luxon.Duration  const dt = DateTime.now();  const dur = Duration.fromObject({ hours: 3, minutes: 15 });  console.log( dur.hours ) // 3  console.log( dur.minutes ) // 15  console.log( dur.seconds ) // 0  console.log( dt.toLocaleString(DateTime.DATETIME\_SHORT) )  // 25/1/2022 14:35  const suma = dt.plus(dur)  console.log(suma.toLocaleString(DateTime.DATETIME\_SHORT))  // 25/1/2022 17:50  const resta = dt.minus(dur)  console.log(resta.toLocaleString(DateTime.DATETIME\_SHORT))  // 25/1/2022 11:20 |
| --- |

**Interval**

De forma similar podemos trabajar la clase Interval, que con su constructor podemos enviar dos fechas y nos devuelve el intervalo entre ellas, pudiendo acceder para realizar los cálculos que necesitemos.

| const Interval = luxon.Interval  const now = DateTime.now()  const later = DateTime.local(2022, 1, 26)  const i = Interval.fromDateTimes(now, later)  console.log( i.length('days') ) // 0.3905736111111111  console.log( i.length('hours') ) // 9.373766666666667  console.log( i.length('minutes') ) // 562.426 |
| --- |