

Enlaces web de interés

-  **Mozilla Developers** - <https://developer.mozilla.org/es/docs/Web/Events>
(website para desarrolladores)
-  **Lenguaje JS** - <https://lenguajejs.com/javascript/>
(documentación sobre JavaScript y su evolución)
-  **JavaScript Info** - <https://es.javascript.info/>
(aprendizaje de JavaScript a través de tutoriales)
-  **freeCodeCamp** - <https://www.freecodecamp.org/>
(comunidad para el aprendizaje de la programación)
-  **ECMAScript 2022** - <https://262.ecma-international.org/13.0/>
(última actualización de la especificación del lenguaje)
-  **W3Resource** - <https://www.w3resource.com/>
(recursos libres para programadores front-end)
-  **DEV Community** - <https://dev.to/>
(comunidad de programadores)



Errores, módulos y API

Objetivos

- Reconocer la importancia de establecer un sistema de gestión de errores.
- Distinguir los diferentes tipos de errores que puede lanzar una aplicación.
- Conocer cómo preparar y lanzar errores.
- Aprender las estructuras que permiten capturar y gestionar errores.
- Descubrir las ventajas de reutilizar código mediante módulos.
- Identificar los problemas de la importación de módulos de terceros.
- Dominar la creación, exportación e importación de módulos propios.
- Asimilar la utilidad de las API.
- Entender el funcionamiento general de una API.
- Estudiar el catálogo de API propias de JavaScript.
- Utilizar algunas API importantes para interiorizar mecánicas de uso.

Contenidos

- 8.1. Errores
- 8.2. Módulos
- 8.3. API

Introducción

Alrededor del proceso de desarrollo de aplicaciones web existe todo un ecosistema de utilidades, más allá del funcionamiento básico de JavaScript, que resultan de gran interés para alcanzar un producto de calidad.

Tener controlados los posibles errores que puede lanzar el programa mediante eficientes gestores de errores puede marcar la diferencia entre una aplicación usable y otra que esté constantemente rompiéndose. Los errores se van a producir siempre, porque entran en juego múltiples factores externos que escapan al control del programador. La clave no está tanto en evitarlos, que también, como en recuperar el sistema correctamente cuando se producen.

Otro elemento que ayuda mucho en el desarrollo de aplicaciones web es el concepto de la reutilización. Trabajar con módulos que se programan una vez y se utilizan en muchos proyectos tiene grandes beneficios, como acortar los tiempos de desarrollo, abaratar el coste del producto o reducir la cantidad de errores.

Por último, conocer y saber utilizar las docenas de API disponibles tanto en el propio lenguaje como en servicios externos aportan a las aplicaciones una funcionalidad de extrema utilidad que muchos desconocen.

Por todo lo anterior, se anima al lector a conocer en esta unidad estos tres ámbitos del desarrollo de aplicaciones web, a practicarlos y a incorporarlos a su catálogo de habilidades porque a buen seguro le reportarán importantes ventajas.

8.1. Errores

Al desarrollar aplicaciones web debe tenerse siempre en mente que el objetivo es alcanzar un producto de calidad. Esto implica que la ejecución sea eficiente, que los tiempos de espera sean cortos, que la aplicación sea usable y amigable, y una larga lista de otras características deseables. Una de estas características, que no siempre se trata con esmero, es la tolerancia a fallos. Es verdad que el trabajo debe realizarse tratando de evitar todos los posibles errores que se puedan generar, pero también es muy importante gestionar correctamente la recuperación de un error cuando este se produce. En otras palabras, preparar el programa para que sea capaz de recuperarse de ciertos errores que no van a poder evitarse.

8.1.1. Qué se entiende por error

Cuando se habla de errores, se está haciendo referencia a todos aquellos comportamientos anómalos de la aplicación que no estaban previstos. Siguiendo con este marco conceptual, un error no es solo el típico mensaje en rojo que suele aparecer en muchas aplicaciones cuando se produce un fallo.

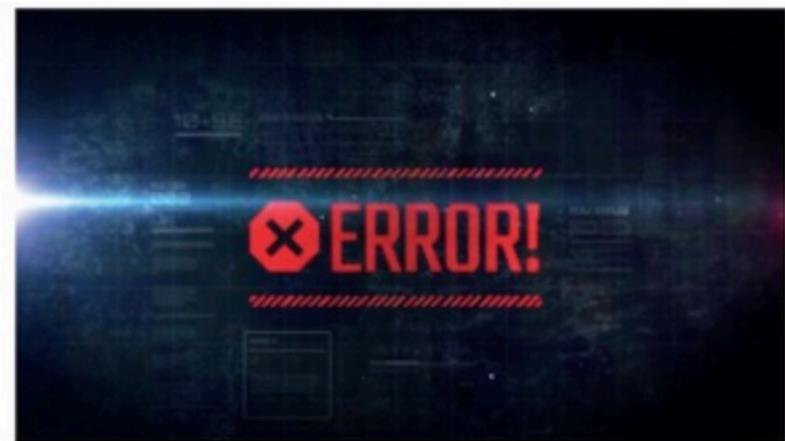


Figura 8.1. Los errores no siempre tienen el aspecto que todos imaginan.

Un error también es que la aplicación realice una operación de una forma no esperada, aunque los resultados aparentemente no sean incorrectos. Estos, como se podrá entender, son los más difíciles de detectar, por eso hay que ser muy cuidadoso y escribir los programas siguiendo una disciplina de comprobación de errores regular.

Todos los desarrolladores invertimos una cantidad enorme de tiempo tratando de encontrar errores. Sin embargo, cuando ya se adquiere cierta experiencia, los programadores desarrollamos un sentido especial para «intuir» en qué partes del código se podría estar produciendo un error. Pero esto no se puede enseñar, simplemente ocurre tras desarrollar un buen número de proyectos.

8.1.2. Tipos de errores

A grandes rasgos, y atendiendo al momento en el que se producen, los errores se pueden clasificar en dos grandes tipos:

- **Errores en tiempo de desarrollo:** son los que se cometan por escribir mal una instrucción. Normalmente el editor de código los detecta y proporciona información sobre qué podría estar ocurriendo. En otras ocasiones, es más complicado porque los errores se encadenan y aquello que muestra el editor es un error heredado del auténtico origen.
- **Errores en tiempo de ejecución:** son los que se producen una vez que la aplicación se está ejecutando. Podría deberse a identificadores no definidos previamente, operaciones aritméticas incorrectas, conexiones no disponibles, interbloqueos, etcétera.

Pensándolo bien, casi todos los errores anteriores son fruto de una mala práctica o de un descuido por parte del programador. Sin embargo, no siempre es así, a veces los errores son inducidos por fallos en el sistema (no hay conexión de red, por ejemplo) o por un mal uso de la aplicación por parte de los usuarios (escribir un DNI en el campo donde se le pide su correo electrónico, por ejemplo). Pero hasta en esos casos, el programador es responsable de tener preparada su aplicación para tolerar esos fallos. Es fundamental que el impacto en el funcionamiento de la aplicación sea mínimo.

Además, no todos los comportamientos anómalos de la aplicación tienen las mismas consecuencias. En unos casos la aplicación deja de funcionar y, en otros, la ejecución continúa, pero con datos inconsistentes. Por eso, se diferencian tres tipos de errores que podrían darse en las aplicaciones web:

- **Error:** son los errores que provocan que la aplicación deje de funcionar. Se considera que este tipo es un error no controlado, que desencadena una serie de fallos que se van acumulando hasta que la aplicación no puede más y, finalmente, se detiene.
- **Exception:** una excepción es un error para el que se ha preparado un gestor que lo administra. Se sabe que puede darse en ciertas situaciones y, en vez de evitarla, se prepara su gestión para que el impacto esté controlado y no afecte significativamente a la ejecución de la aplicación.
- **Warning:** los avisos son los errores menos importantes. No provocan que la aplicación deje de funcionar, pero notifican que se están produciendo situaciones preocupantes que podrían terminar generando un error fatal. A pesar de no ser de gran importancia, deben tenerse en cuenta e interesarse por saber qué está ocurriendo para intervenir y que la situación no empeore.

8.1.3. Creación y lanzamiento de errores

JavaScript dispone de toda una jerarquía de objetos para gestionar de forma automática la información que proporciona sobre ellos y también para que los programadores puedan desarrollar sus propias estrategias de control de errores.

Los errores en tiempo de ejecución dan como resultado la creación y lanzamiento automático de objetos **Error**. Pero también se puede usar esta infraestructura de objetos para crear y lanzar errores propios:

```
let error = new Error("Formato de fecha incorrecto");
```

La instrucción anterior en realidad no lanza ningún error, sino que simplemente crea el objeto que se usará para almacenar toda la información sobre él. Para lanzar un error se utiliza **throw**:

```
let error = new Error("Formato de fecha incorrecto");
throw error;
```

Ahora sí que se obtiene un llamativo aviso de que algo no ha ido bien (Figura 8.2).



Figura 8.2. La consola muestra un error personalizado por el programador.

El error creado es un error genérico, pero pueden definirse errores de tipos mucho más específicos, tal y como permite JavaScript con los siete constructores que se recogen en la Tabla 8.1.

Tabla 8.1. Constructores específicos para tipos de errores concretos

Constructor	Utilidad
EvalError	Error lanzado al utilizar la función eval() , que evalúa una expresión y retorna su valor.
InternalError*	Error interno en el motor de JavaScript, por ejemplo: demasiada recursividad, demasiados paréntesis en una expresión regular, demasiadas opciones en un switch...
RangeError	Error lanzado cuando una variable numérica o parámetro toma un valor que está fuera de sus valores establecidos.
ReferenceError	Error que se produce cuando se realiza una referencia a objeto no válida.
SyntaxError	Error producido cuando JavaScript intenta interpretar código que no es válido sintácticamente.
TypeError	Error lanzado cuando el tipo de una variable o parámetro no es válido.
URIError	Error que se produce al codificar encodeURI() o al decodificar decodeURI() un URI.

* Es un constructor no estándar. No se debe usar en sitios web en producción porque no funcionará a todos los usuarios.

Existen también dos propiedades estándar (ciertos desarrolladores de navegadores como Mozilla o Microsoft proporcionan muchas otras) para acceder a información concreta del error, como el mensaje del error (**message**) y su nombre (**name**):

```
let error = new Error("Formato de fecha incorrecto");
console.log('Mensaje del error: ${error.message}');
console.log('Nombre del error: ${error.name}');
```

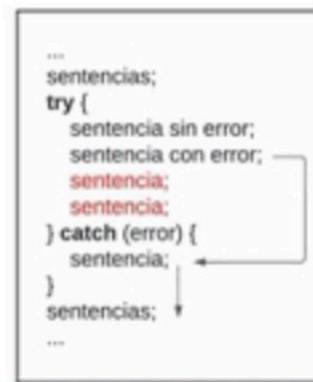
8.1.4. Gestión de excepciones

JavaScript proporciona un mecanismo muy útil para definir aquellos bloques de instrucciones sospechosos de poder generar errores y capturarlos con facilidad. Se trata de la declaración **try...catch**.

En el bloque **try** se colocan las instrucciones que podrían generar un error, y en el bloque **catch** las declaraciones que especifican qué hacer si se llega a producir. A continuación se muestra un ejemplo sencillo de su uso:

```
try {
    throw "Error inducido";
} catch (error) {
    console.log(error);
}
```

Si alguna instrucción dentro del bloque **try** lanza una excepción, el control del programa salta automáticamente al bloque **catch**. Cuando termine la ejecución del bloque **catch**, no se seguirán ejecutando las instrucciones que quedaron pendientes del bloque **try**.

Figura 8.3. Lógica de ejecución de un bloque `try...catch`.

Actividad resuelta 8.1

Fuera de rango

Escribe un programa que compruebe el valor de una variable numérica entera llamada `nota` y lance un error si su valor no se encuentra entre 0 y 10.

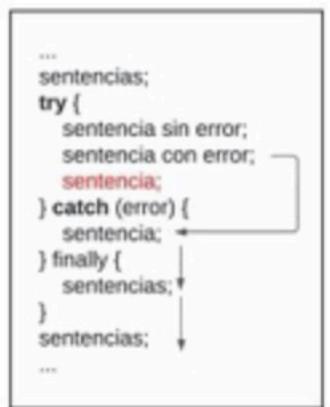
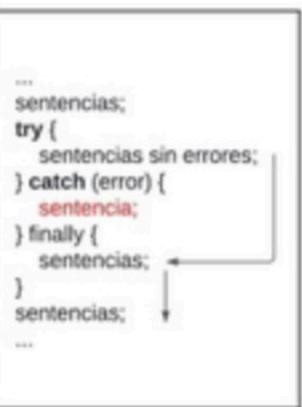
Solución

```

function compruebaNota(n) {
    if (n<0 || n>10) {
        throw new RangeError("Valor incorrecto para la nota.");
    }
}
try {
    compruebaNota(21);
} catch (error) {
    if (error instanceof RangeError) {
        console.log(`${error.name}: ${error.message}`);
        console.log("Por favor, vuelve a introducir un valor para la nota.");
    }
}
  
```

Existe otra variante a esta declaración que es la estructura `try...catch...finally`. En este caso, el bloque `finally` contiene instrucciones que se ejecutarán después de que se ejecuten los bloques `try` y `catch`. Es crucial entender que el bloque `finally` se ejecutará siempre, tanto si se ha lanzado un error como si no. Incluso, si se lanza un error, el bloque `finally` se ejecutará, aunque el bloque `catch` no haya capturado el error concreto que se lanzó.

Otro detalle muy importante de `finally` es que su valor devuelto es el de referencia para todo el bloque `try...catch...finally`. Esto quiere decir que, si tanto el bloque `try` como el bloque `catch` contienen instrucciones `return`, estas quedarán invalidadas mientras no se termine de ejecutar el bloque `finally`, cuyo `return` será el único que retorne el auténtico valor de toda la estructura, puesto que sobrescribe a los anteriores. De la misma manera, esta sobreescritura también se aplica a las excepciones lanzadas o relanzadas en el bloque `catch`.

Figura 8.4. Lógica de ejecución de un bloque `try...catch...finally` con error en el bloque `try`.Figura 8.5. Lógica de ejecución de un bloque `try...catch...finally` sin error en el bloque `try`.

¿Qué utilidad tiene entonces el bloque `finally`? Mucha; por ejemplo, liberar un recurso que el programa haya bloqueado: puede tratarse de una aplicación que abre un archivo y realiza una serie de operaciones con él. Si se produce un error, se podría usar `finally` para cerrar el archivo y que esté disponible para el resto de la aplicación. De esta manera, se evita que el programa se detenga.

Actividad propuesta 8.1

Errores múltiples

Crea un fragmento de código que lance al menos tres tipos de errores distintos. Se pide capturar y gestionar los tres tipos de errores (mostrándolos en la consola) con una sola declaración `try...catch`.

8.1.5. El modo estricto

Siempre se ha hecho referencia a JavaScript como un lenguaje que dispone de una sintaxis muy flexible. Su naturaleza de lenguaje débilmente tipado hace que no necesite conocer al detalle con qué tipo de dato se está trabajando en cada momento, tal y como se estudió en la Unidad 2. Sin embargo, estas características no son del agrado de muchos programadores, que prefieren un entorno más estricto que puedan controlar y no dejar en manos del intérprete ciertas libertades que podrían causar inconsistencias. Esta circunstancia, unida al auge del desarrollo de aplicaciones con JavaScript cada vez más grandes, con más consumo de recursos y también incorporando el lado servidor, hace que el control de errores cada vez sea más importante y, en ocasiones, sea más seguro utilizar lo que se conoce como el modo estricto. Se trata de un modo más exigente de tratar los errores que fue incorporado en la versión ES5 del estándar.

Para activar el modo estricto se escribe '`use strict`'; (comillas incluidas) en la primera línea del fichero de código JavaScript. Si no se desea activar para todo el fichero puede hacerse de forma selectiva, ya que también es posible circunscribirlo al ámbito de una

función, de manera que solo se aplicarán las normas que incorpora en el interior de la función. A continuación se presenta un ejemplo muy simple.

Si el programa contiene esta única instrucción:

```
cadena = "una cadena de caracteres";
```

no ocurrirá nada, porque JavaScript automáticamente habrá asignado el tipo a la variable de forma dinámica. Sin embargo, si se escribe:

```
'use strict';
cadena = "una cadena de caracteres";
```

el navegador mostrará un mensaje de error indicando que la variable no está definida (Figura 8.6).



Figura 8.6. El modo estricto detectando errores de tipos.

8.2. Módulos

Un módulo en JavaScript es un conjunto de objetos, funciones, constantes, etc., que puede usarse como una librería y que puede reutilizarse para agilizar el desarrollo de aplicaciones.

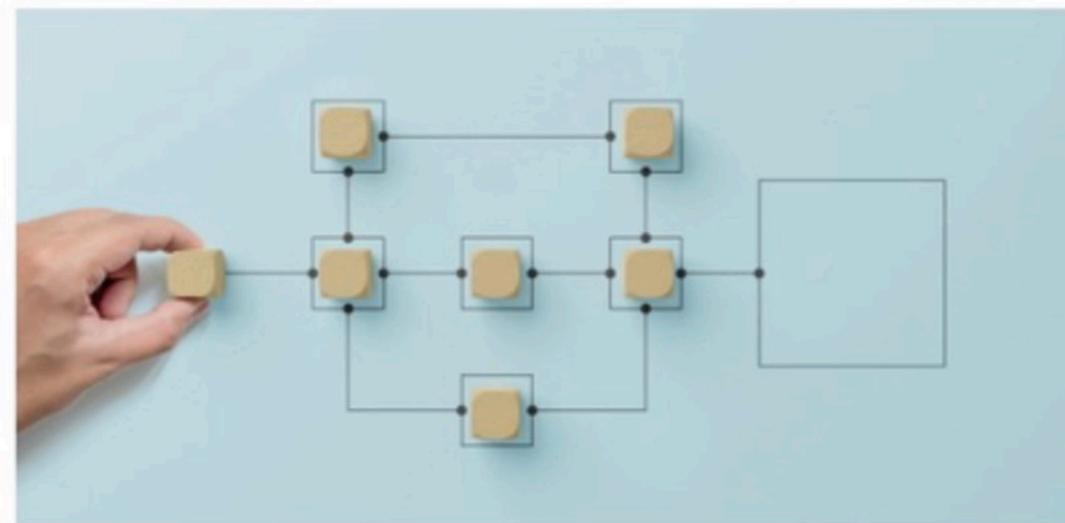


Figura 8.7. El uso de módulos facilita el desarrollo de aplicaciones combinando componentes.

La necesidad de utilizar módulos con JavaScript del lado cliente ha sido una reivindicación tradicional entre la comunidad de programadores desde hace mucho tiempo. Pero

en la actualidad ha subido de intensidad debido a la fácil gestión que, por ejemplo, realiza Node.js de los módulos, una tarea completamente integrada en su gestor de paquetes npm.

Además, el uso de módulos no solo mejora la reutilización del código, sino que permite tener aplicaciones más rápidas y eficientes. En muchas ocasiones, se necesita incorporar a las aplicaciones librerías de terceros de las que se desean utilizar muchas menos utilidades de las que proporciona. Sin embargo, hay que descargar todo el fichero para poder usar la pequeña parte que se necesita. Y esto es un desperdicio innecesario de recursos.

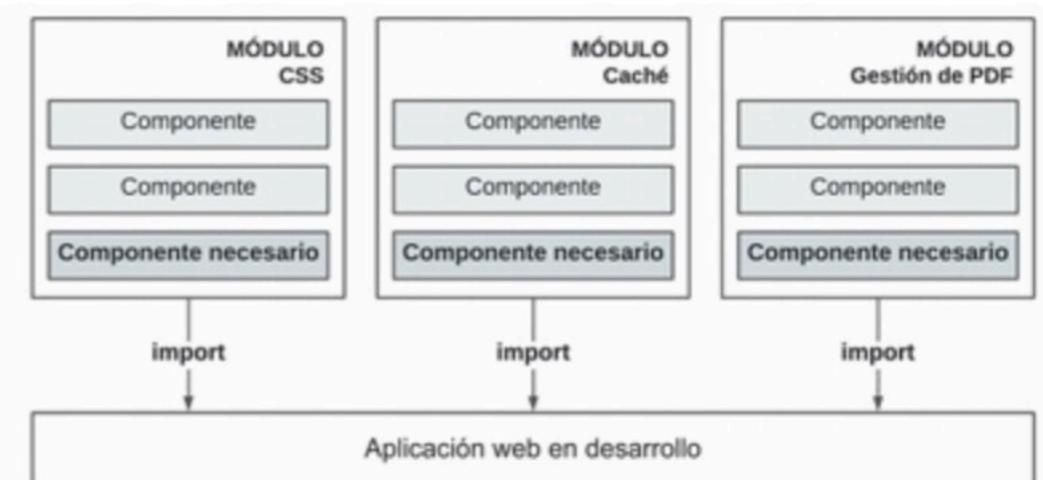


Figura 8.8. Descarga innecesaria de módulos para usar una pequeña parte de ellos.

Es verdad que existen algunas herramientas que pueden utilizarse para realizar esa carga selectiva que se necesita, como por ejemplo la archiconocida **Require.js**. Pero vuelve a surgir el problema de que no forma parte del estándar y, por tanto, se siguen desarrollando aplicaciones sin garantías de compatibilidad en la carga de módulos.

Lo que sí puede hacerse, desde la incorporación de **export** e **import** al estándar ES6, es crear y cargar módulos propios.

Para **crear un módulo** se marca cada una de las utilidades que se quieren importar más tarde con la palabra clave **export**. Por ejemplo, si se tiene un módulo llamado **operar_cadenas.js** se podría hacer lo siguiente:

```
function esPalindroma(cadena) {
    // sentencias;
}
export function anagramas(leyendas) {
    // sentencias
}
export function inversa(cadena) {
    // sentencias
}
```

O también se puede recurrir a la opción de agrupar al final aquello que se desea exportar:

```
function esPalindroma(cadena) {
    // sentencias;
}
function anagramas(letras) {
    // sentencias
}
function inversa(cadena) {
    // sentencias
}
export {
    anagramas,
    inversa
}
```

De esta manera, los elementos que se han exportado pueden ser importados por otro programa. El resto, quedará como privado.

Para **importar** una o varias utilidades de un módulo no hay más que indicarlo de esta manera:

```
import { anagramas, inversa } from "./operar_cadenas.js";
let miArray = anagramas("girasoles");
let invertida = inversa("abad");
```

La importación también permite **crear alias de los elementos importados**, de manera que si se escribe `import {anagramas as ana} from "./operar_cadenas.js"` se puede invocar a la función `anagramas` usando el identificador `ana`.

También existe la posibilidad de **utilizar espacios de nombres** al importar todos los elementos de un módulo. Si se hace esto:

```
import * as opca from "operar_cadenas.js";
let miArray = opca.anagramas("girasoles");
```

debe usarse el espacio de nombres cada vez que se invoque a un elemento importado. ¿Qué utilidad tiene esto? Una muy importante: diferenciar con un golpe de vista qué módulo proporciona un elemento que se está usando, cuando se han importado varios módulos. Y otra más importante aún, evitar colisiones de nombres cuando dos o más módulos contienen elementos con el mismo nombre.

Un último detalle muy importante que hay que tener en cuenta es que al importar módulos, si se escribe el código JavaScript en el contenido de la etiqueta `script` de HTML, debe indicarse `type="module"`, en lugar de `type="text/javascript"`.

Actividad propuesta 8.2

Biblioteca matemática

Crea un módulo que contenga dos constantes matemáticas y tres funciones que consideres útiles. Solo debes exportar una constante y dos funciones. Luego escribe un programa que haga uso del módulo, importando solo aquellos elementos que vayas a usar.

8.3. API

API son las siglas de *Application Programming Interfaces* o interfaz de programación de aplicaciones. Se trata de un conjunto de objetos, propiedades, métodos, funciones y otros componentes que se utilizan para desarrollar aplicaciones web permitiendo la comunicación entre dos aplicaciones siguiendo un conjunto de reglas. Pueden entenderse como un módulo externo de software que se comunica o interactúa con otro para conseguir ciertos objetivos. Tomando como suposición el desarrollo de una aplicación web que necesita mostrar la información meteorológica de ciertas localizaciones, en vez de programar a mano la lectura de sensores de una estación meteorológica de cada una de las localizaciones, lo que se hace es buscar un servicio *online* que ya tenga implementado este trabajo, y se realizan peticiones a través de su API para que la aplicación se encargue simplemente de mostrar los datos que recibe.

Desde el punto de vista del usuario esta comunicación es completamente transparente, el usuario ve en la aplicación la información meteorológica, pero no sabe cómo ni desde dónde se están consumiendo los datos.

Al registrarse o iniciar sesión en muchos sitios web, se pueden utilizar las credenciales propias de Google, Twitter o Facebook. Lo que está haciendo la aplicación web en la que se quiere iniciar sesión es conectarse en segundo plano con las API de esos servicios para identificar al usuario, sin necesidad de tener que abrir un largo y tedioso proceso de registro que incomoda a muchos.

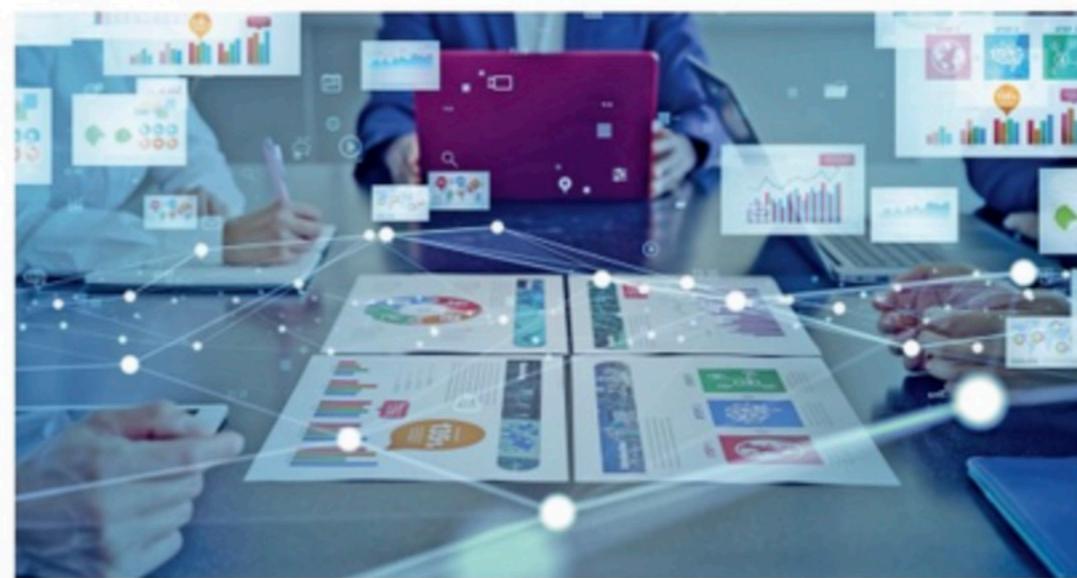


Figura 8.9. Consumir componentes internos o externos mediante API abre múltiples posibilidades.

A lo largo del aprendizaje de JavaScript se ha hecho uso de alguna API; ¿qué es, si no, el propio DOM? Una API que ofrece objetos, como `window` o `document`, y propiedades y métodos que permiten gestionar los elementos del DOM. En la siguiente relación se recopilan algunos otros ejemplos de API que HTML5 proporciona:

- **API Canvas:** ofrece funcionalidad para dibujar en 2D en un lienzo de HTML5.
- **Application Cache:** muy útil para que las aplicaciones web puedan trabajar *offline*.
- **Battery Status:** información detallada de la batería del dispositivo.
- **Drag & Drop:** amplia funcionalidad para trabajar con elementos arrastrables.
- **FileSystem API:** trabajo con archivos locales usando JavaScript.
- **FullScreen:** utilidades para poder gestionar la aplicación cuando se visualiza a pantalla completa.
- **Geolocation:** gestión de las coordenadas de localización del usuario.
- **Media API:** funcionalidad extendida para gestionar contenido de audio y vídeo.
- **Text Track API:** maneja los subtítulos de componentes de audio y vídeo.
- **Web GL:** amplía la funcionalidad del canvas para trabajar con 3D incorporando elementos de Open GL.
- **Web Sockets:** gestiona los sockets de una red para facilitar la comunicación cliente-servidor.
- **Web Storage:** permite esquivar las *cookies* almacenando datos en el navegador.
- **Web Workers:** ofrece la posibilidad de realizar procesamiento en segundo plano para evitar problemas de rendimiento en el primer plano.

Actividad resuelta 8.2

Dibujo en 2D

Crea un programa, usando la API Canvas, que disponga de un lienzo de 500 x 500 px y solo utilizando código JavaScript consiga pintar la cara de un emoticono sonriendo.

Solución

HTML

```
<canvas id="lienzo" width="500" height="500"></canvas>
<script type="text/javascript" src="errores-modulos-apis.js"></script>
```

JavaScript

```
function dibujarCarita() {
    let lienzo = document.getElementById("lienzo");
    if (lienzo.getContext){
        let mano = lienzo.getContext("2d");
        mano.beginPath();
        mano.arc(75,75,50,0,Math.PI*2,true);
        mano.moveTo(110,75);
        mano.arc(75,75,35,0,Math.PI,false);
        mano.moveTo(65,65);
        mano.arc(60,65,5,0,Math.PI*2,true);
        mano.moveTo(95,65);
        mano.arc(90,65,5,0,Math.PI*2,true)
        mano.stroke();
    }
    dibujarCarita();
}
```

A continuación se repasará brevemente la mecánica de uso de tres de las API más usadas en la actualidad: almacenamiento, geolocalización y notificaciones.

Actividad propuesta 8.3

Más API

Descubre alguna otra API propia de JavaScript que no hayamos citado aquí y escribe un programa donde expliques su funcionamiento básico.

8.3.1. API Web Storage

Web Storage permite almacenar datos en la máquina del usuario. La gestión de datos que ofrece esta API es mucho más interesante que en el caso de las *cookies* (aunque en casos muy concretos las *cookies* son necesarias). Por citar solo algunas ventajas:

- Menos restricciones que las *cookies*.
- Los datos no se envían al servidor.
- No se incluyen en ninguna petición/respuesta HTTP.
- Permiten almacenar varios gigabytes.

Desde el punto de vista del código, el objeto `window` ofrece dos nuevas propiedades: `localStorage`, para almacenar los datos; y `sessionStorage`, que almacena datos con gestión de la sesión. La única diferencia entre las dos es que los datos almacenados con `sessionStorage` se eliminan tan pronto como acaba la sesión.

Nota técnica

Es importante saber que para usar Web Storage con este tipo de datos es necesario que las páginas se carguen desde un servidor web. No funcionará si se abre directamente el fichero en el navegador. Para ello, puede instalarse un servidor web tipo XAMP/LAMP o instalar algún plugin en el editor de código del tipo Live Server (Visual Studio).



En los siguientes ejemplos se muestra cómo realizar algunas operaciones sencillas usando esta API.

Para almacenar un dato se procede igual que con las *cookies*, indicando una clave y un valor por medio del método `setItem`:

```
localStorage.setItem("unaClave","Un valor");
```

Para recuperar un dato almacenado se utiliza `getItem` o se usa directamente una clave:

```
localStorage.getItem("unaClave");
localStorage.unaClave;
```

Además, como siempre que se trabaja con JavaScript, se puede recurrir al objeto **JSON** para guardar estructuras de datos más complejas usando los ya conocidos métodos **stringify** y **parse**:

```
localStorage.setItem("unaClave",JSON.stringify(miCadenaJSON));
miObjetoJSON = JSON.parse(localStorage.getItem("unaClave"));
```

Igualmente, para eliminar una clave almacenada y su valor asociado se recurre a **removeItem** o el operador **delete**:

```
localStorage.removeItem("unaClave");
delete localStorage.unaClave;
```

Y si lo que se quiere es liberar toda la memoria ocupada limpiando todo el almacenaje, se acude a **clear**:

```
localStorage.clear();
```

8.3.2. API Geolocation

Como se puede deducir, esta API permite conocer cierta información geográfica del dispositivo del usuario.

Las aplicaciones son tan extensas como alcance la imaginación del programador. Mientras sea necesario ofrecer o consumir información del entorno más cercano a la posición del usuario, más sentido cobrará el uso de esta API. Lógicamente, la posición del dispositivo de un usuario forma parte de su privacidad, por lo que la aplicación solicitará permiso para conocer su ubicación.

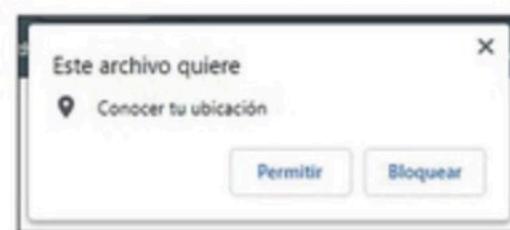


Figura 8.10. Solicitud de permiso de ubicación al usuario en Chrome.

En este caso, las utilidades proporcionadas por esta API se encuentran en el objeto **navigator** cuyo estudio se adelantó al ver el DOM en la Unidad 6.

Para conocer la posición del usuario se tiene que invocar a **getCurrentPosition**, cuyo parámetro es un **callback** que recibe un objeto **Position**. Es este último objeto el que da acceso a la longitud y latitud del dispositivo. Se haría lo siguiente:

```
navigator.geolocation.getCurrentPosition(posicion=>
  console.log('Lon: ${posicion.coords.longitude}');
  console.log('Lat: ${posicion.coords.latitude}');
});
```

Y si se acepta que el fichero conozca la ubicación, se mostrarían los datos.

Pero estas propiedades no son las únicas que ofrece el objeto **Position**. Dispone de algunas otras también muy interesantes, como las que se muestran en la Tabla 8.2.

Tabla 8.2. Algunas propiedades del objeto Position

Propiedad	Utilidad
accuracy	Precisión del cálculo de la longitud y la latitud (en metros).
altitude	Altura sobre el nivel del mar.
altitudeAccuracy	Precisión del cálculo de la altitud (en metros).
heading	Ángulo de movimiento del dispositivo del usuario.
speed	Velocidad de movimiento del dispositivo del usuario (en m/s).

Otro método enormemente útil es **watchPosition** que funciona de forma muy parecida al anterior, pero se ejecuta constantemente para conocer variaciones de la posición y poder hacer un seguimiento del dispositivo. ¿Cuándo deja de consultar la posición? Cuando se invoca al método **clearWatch**.

Por ejemplo, si solo se quiere rastrear la posición del usuario cuando su posición supera los 1000 metros de altitud, se podría hacer esto:

```
let identificador = navigator.geolocation.watchPosition(posicion=>{
  console.log('Lon: ${posicion.coords.longitude}');
  console.log('Lat: ${posicion.coords.latitude}');
  if (posicion.coords.altitude < 1000)
    navigator.geolocation.clearWatch(identificador);
});
```

Este programa iría sacando en consola la posición del dispositivo una y otra vez mientras su altitud esté por encima de los 1000 metros. Utilizando el identificador de seguimiento devuelto por **watchPosition** se puede interrumpir el seguimiento y centrarse solo en aquella cota que interesa.

8.3.3. API Notification

La democratización del acceso a los dispositivos móviles ha tenido muchas consecuencias para la mayoría de los lenguajes de programación, y JavaScript no es una excepción. Desde bien pronto las aplicaciones móviles implementaron un completo sistema de notificaciones que con el tiempo han forzado a tecnologías como JavaScript a incorporarlas en su catálogo de API. En este caso el objeto que las gestiona se llama **Notification**.

La notificación es otra funcionalidad intrusiva para el usuario, por lo que también debe solicitarse su permiso. Para ello se utiliza un método estático llamado **requestPermission()** que proporciona una promesa resuelta positivamente cuando el usuario contesta a la petición (la respuesta será "granted" si se aceptó o "denied" si se rechazó el permiso).

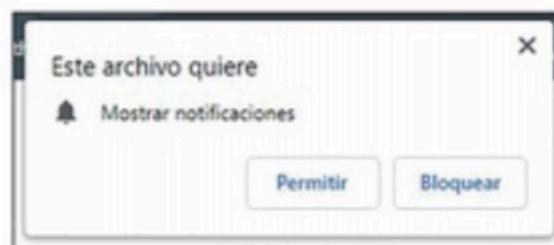


Figura 8.11. Solicitud de permiso para enviar notificaciones al usuario en Chrome.

Esta solicitud no hay que hacerla siempre que vaya a enviarse una notificación al usuario. Si el valor de `Notification.permission` al consultarla es `default`, quiere decir que aún no se le ha solicitado permiso, con lo cual hay que pedirlo. No es recomendable realizar la solicitud de permiso más de una vez por sesión. Además, incluso habiendo recibido el permiso del usuario, no debe abusarse del sistema de notificaciones, ya que suele generar un enorme rechazo entre los usuarios.

Una vez obtenido el permiso, enviar una notificación es tan sencillo como crear un objeto `Notification`:

```
Notification.requestPermission().then(respuesta=>{
    if (respuesta == "granted")
        new Notification("¡Gracias!");
});
```

Este mensaje aparecerá en el área de notificaciones específica de cada sistema operativo.

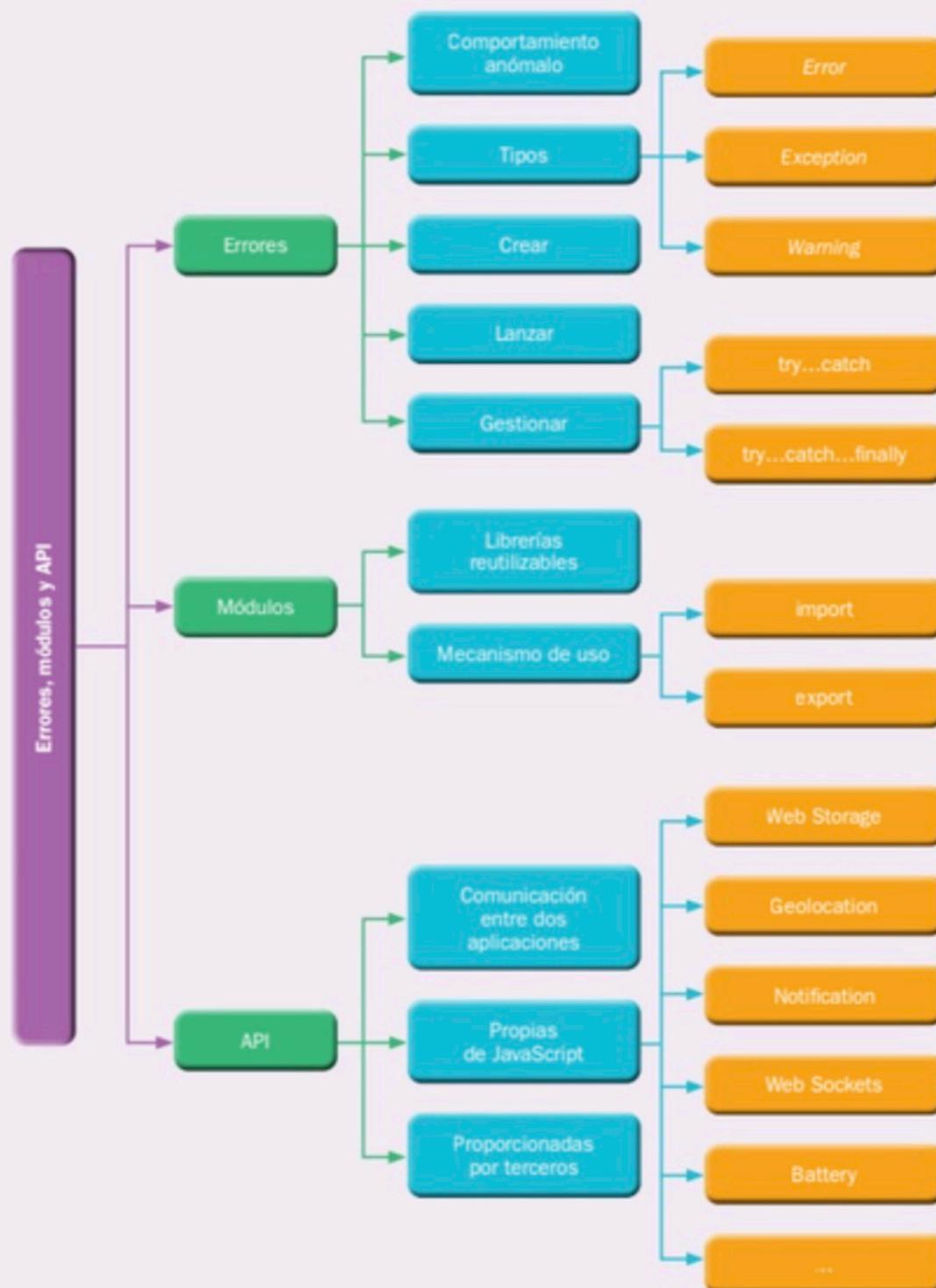
Las notificaciones se pueden crear tan avanzadas como se quiera: incorporar un icono propio a la notificación, darle formato a la notificación usando el segundo parámetro del constructor, cerrarla manualmente con el método `close()`, asociarle eventos como abrir una URL tras un clic, etcétera.

Cuando se utilizan las notificaciones muy dosificadas y con información útil, son una herramienta excepcional para interaccionar con el usuario. Cuando se abusa o se notifican informaciones no relevantes para el usuario, la aplicación web está condenada al rechazo.

Para saber más

Con este enlace o con el código QR se accede a la lista completa de API que proporciona JavaScript con enlaces a su especificación, lista de propiedades, métodos y ejemplos de uso. El repaso de esta referencia puede cambiar completamente la percepción de hasta dónde llegan las capacidades de JavaScript. Es altamente recomendable revisarla:

<https://developer.mozilla.org/en-US/docs/Web/API>



Algunas líneas atrás se citó un concepto nuevo llamado **promesa**, y en el código se puede ver un método llamado `then` que tampoco se había citado con anterioridad en esta obra. Se llega casi sin querer al último gran escalón que quedaba por subir: la programación asíncrona con JavaScript.