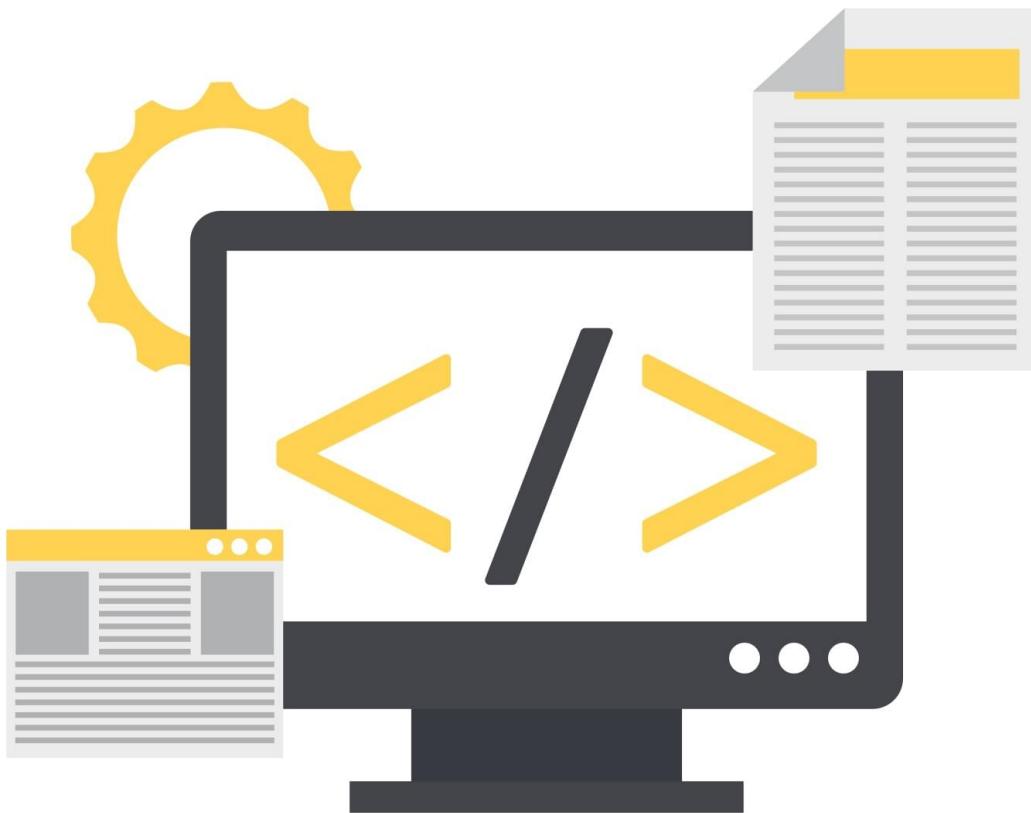


JavaScript

Aprende a programar
en el lenguaje de la Web



- ⌚ Instalar el entorno de trabajo
- ⌚ Sintaxis, arreglos, objetos, métodos
- ⌚ Manejo del DOM y componentes HTML
- ⌚ Programación para smartphones

USERS

USERS

Servicio Técnico de PCs

Servicio Técnico de PCs

GU06



- ⌚ Armar una PC
- ⌚ Reemplazo de componentes
- ⌚ Diagnóstico y reparación
- ⌚ Hardware Stressing

Javier Richarte

PROCEDIMIENTOS DE REPARACIÓN PASO A PASO

PROXIMO TITULO

CONSEGUILO EN KIOSCOS Y LIBRERIAS DE TODO EL PAÍS
O SUSCRIBIENDOTE EN USERSHOP.REDUSERS.COM



+54-11-4110-8700



usershop@redusers.com

JavaScript

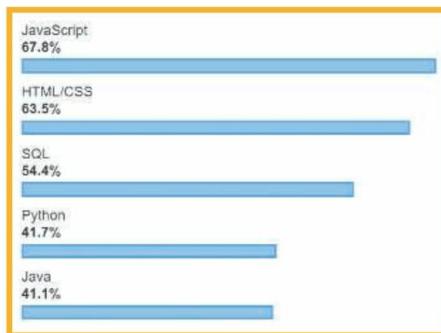
Aprende a programar en el lenguaje de la Web

JavaScript es el lenguaje de programación más utilizado hoy. Junto a HTML y CSS, le da vida a la gran mayoría de los sitios Web que visitamos. Pero JavaScript ya no es exclusivo de la Web: proyectos como **NodeJS**, **Electron** y **React Native** lo llevaron al ámbito de los servidores, los programas de escritorio y las aplicaciones móviles. Además, cada día surgen nuevos **frameworks** (como los líderes y archienemigos **React** y **Angular**) para expandir las posibilidades de JavaScript.

En esta guía introductoria, Fernando Luna, con su larga experiencia de docente y technical writer, nos llevará en un viaje vertiginoso desde el primer "Hola Mundo" hasta aplicaciones que nos hablan y acceden a bases de datos. Obviamente, este no pretende ser un curso comprehensivo, sino un punto de partida para que el lector comience a programar en este excitante lenguaje.

Miguel Lederkremer (@leder)
Director Editorial

Las 5 tecnologías más populares del momento en Stack Overflow, el foro de programación líder.



facebook.com/redusers



twitter.com/redusers



instagram.com/reduserscom/



redaccion@redusers.com

Luna, Fernando O.

JavaScript: aprende a programar en el lenguaje de la Web / Fernando O. Luna. - 1a ed.- Ciudad Autónoma de Buenos Aires: Six Ediciones, 2019.
144 p. ; 28 x 20 cm.

ISBN 978-987-4958-08-2

1. Lenguaje de Programación. I. Título.

CDD 005.133

TÍTULO

JavaScript / Aprende a programar en el lenguaje de la Web

AUTOR

Fernando O. Luna

EDICIÓN

Claudio Peña

DISEÑO Y PRODUCCIÓN

Gustavo De Matteo

COLECCIÓN Users Guías

FORMATO 28 x 20 cm

PÁGINAS 144

ISBN 978-987-4958-08-2

Copyright © MMXIX. Es una publicación de SIX EDICIONES. Hecho el depósito que marca la ley 11723. Todos los derechos reservados. Esta publicación no puede ser reproducida ni en todo ni en parte, por ningún medio actual o futuro, sin el permiso previo y por escrito de SIX EDICIONES. Su infracción está penada por las leyes 11723 y 25446. La editorial no asume responsabilidad alguna por cualquier consecuencia derivada de la fabricación, funcionamiento y/o utilización de los servicios y productos que se describen y/o analizan. Todas las marcas mencionadas en este libro son propiedad exclusiva de sus respectivos dueños. Impreso en Argentina. Libro de edición argentina. Primera impresión realizada en Casano Gráfica S.A. - Ministro Brin 3932 (1826) R. de Escalada (Lanús) Prov. de Buenos Aires - Argentina, en IV, MMXIX.

JavaScript

Aprende a programar
en el lenguaje de la Web

13

PROYECTOS
EXPLICADOS
PASO A PASO

INTRODUCCIÓN

Qué es JavaScript	4
Configurar el entorno de trabajo.....	6
VSCode	8
JS integrado versus JS independiente	10
JavaScript fuera del HTML	16
Sintaxis básica de JavaScript.....	18
Manejo de condicionales IF-ELSE.....	24

CONCEPTO INICIALES

Introducción a la Programación	
Orientada a Objetos.....	26
Fechas, intervalos y cronómetros	31
Arreglos	38
Cadenas de texto	42
Formularios y datos	48

PROYECTOS

Controlar el DOM HTML	56
Hacer hablar a JavaScript.....	62
Utilizar la API de notificaciones	66
Almacenar datos localmente.....	74
Implementar SQL offline	80
Encriptar contenido	86
Detectar conectividad a Internet.....	92
Adaptar gráficos y multimedia según la performance.....	96
Capturar fotografías y videos.....	104
Acceder al hardware de los dispositivos	112
Los sensores de movimiento.....	120
Manejo de datos remotos con JSon.....	126
Proteger el código mediante ofuscación.....	142



Los archivos de práctica marcados
con este ícono se pueden descargar
gratuitamente desde
redusers.com/u/guiaUsers5



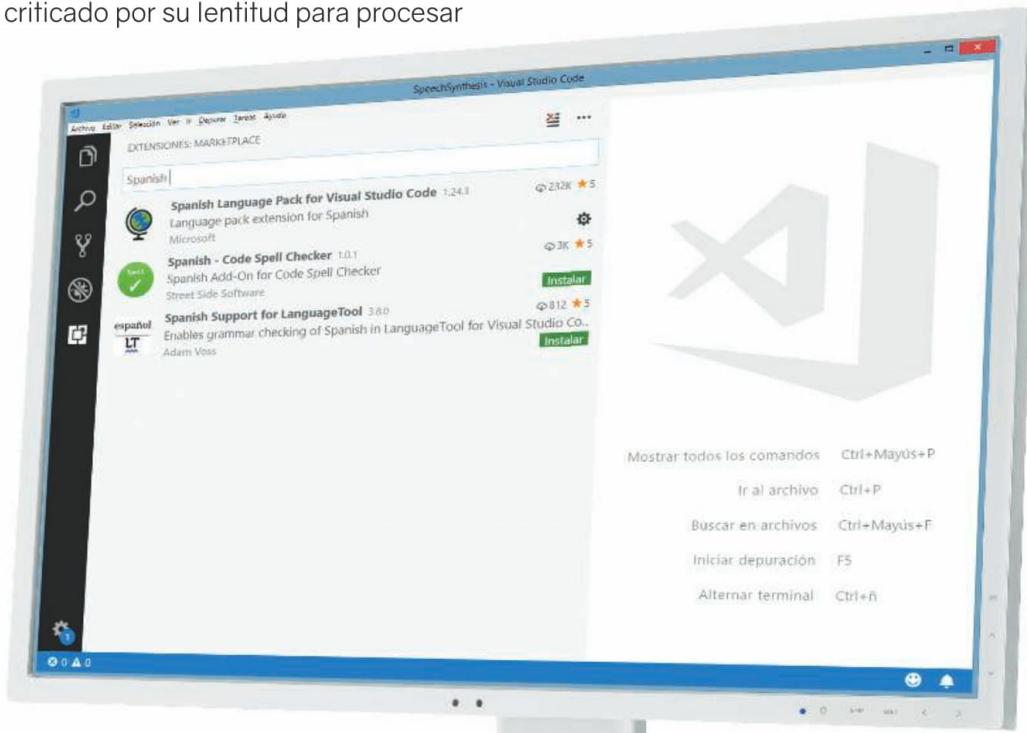
Qué es JavaScript



JavaScript, comúnmente abreviado como JS, es un lenguaje de programación del tipo interpretado. Está basado en el estándar ECMAScript y, a pesar de ser débilmente tipado y dinámico, también se define como un lenguaje orientado a objetos.

JavaScript nació en el año 1995, prácticamente de la mano de la WWW, o Internet comercial. Fue diseñado en un principio por **Netscape Communications** (hoy, **Mozilla Foundation**). Integrado originalmente como una especie de plugin en los navegadores de la primera era web, fue muy criticado por su lentitud para procesar

código, dado que esta acción generaba un retardo notable en la carga completa de una página. A los pocos años de vida fue destronado por **Flash Player**, pero tan solo una década después, volvió completamente recargado, y recuperó su lugar y labor dentro del desarrollo de sitios web.



Sintaxis

JavaScript toma su mayor esencia del lenguaje C, y utiliza convenciones de lenguaje y algunos nombres que son propios de Java, aunque Java y JavaScript no guardan relación alguna entre sí. Las últimas versiones de JS permiten que este sea un lenguaje del lado del cliente, como así también del servidor.

```
// Ejemplo de listas en C
int fibonacci[] = {1, 2, 3, 5, 8, 13, 21, 34};

// Ejemplo de listas en Javascript
var fibonacci[] = {1, 2, 3, 5, 8, 13, 21, 34};

// Otra forma de declarar una lista en Javascript
var fibonacci = new Array(1, 2, 3, 5, 8, 13);
```

Integración en navegadores web

Actualmente, JavaScript está integrado dentro del motor de los navegadores web más populares. Esto determinó diferentes implementaciones del lenguaje, y dependiendo del motor, JS puede tener o no ciertas funcionalidades particulares.

MOTOR	NAVEGADOR WEB
Chakra / Edge HTML / Chromium (desde mediados de 2019)	Internet Explorer / Microsoft Edge (respectivamente)
V8	Google Chrome / Chromium
JavaScript Core	Apple Safari
Rhino	Mozilla Firefox
SpiderMonkey	Mozilla Firefox (hasta su versión 25)
KJS	Proyecto KDE (Konqueror Web Browser)

Los navegadores web más populares y su versión adaptada de JavaScript.

Al momento de escribir esta obra, los mayores motores web del mercado son de **V8**, **Rhino** y **Chakra/EdgeHTML**. Por lo tanto, cualquier desarrollo en JavaScript debe probarse en estos motores, para garantizar su funcionamiento en los demás.



10 ventajas de utilizar JavaScript puro

Existen decenas de frameworks o implementaciones alternativas que utilizan JavaScript como lenguaje y que, a través de diferentes APIs, favorecen el llamado de las sentencias y funciones de este (Jquery, ReactNative y Angular, entre otros tantos).

A pesar de la facilidad que nos dan estos frameworks, hay muchos beneficios que brinda JavaScript como lenguaje puro. Entre ellos, podemos destacar los siguientes:

- 1** Es un lenguaje sencillo y a su vez poderoso
- 2** Cuando está integrado en los motores web, ejecuta rápidamente su sintaxis
- 3** Ya está integrado en los navegadores web más populares
- 4** Es versátil para el desarrollo web dinámico y de aplicaciones móviles
- 5** Es soportado por todos los dispositivos móviles actuales



- 6** Permite el desarrollo de apps móviles híbridas
- 7** Es multiplataforma (computadoras, tablets, móviles, hardware)
- 8** Elimina el peso adicional que un framework genera, disminuyendo el consumo de ancho de banda
- 9** Se relaciona de modo fluido y transparente con HTML y CSS
- 10** Base de los frameworks JS más populares

Configurar el entorno de trabajo



Para poner manos a la obra, delinearemos a continuación nuestro ambiente de desarrollo, instalando las herramientas necesarias destinadas a realizar algoritmos óptimos y probar cada uno de los ejercicios de la forma más real posible.

Para llevar a cabo esta tarea, lo primero que haremos será instalar el IDE donde programaremos todos los ejemplos prácticos. La sigla IDE (*Integrated Development Environment*) significa “entorno de desarrollo integrado”. Se trata de un programa o interfaz que permite

estructurar un proyecto que requiere ser programado. En nuestro caso, el IDE que vamos a utilizar da la posibilidad de incorporar una serie de herramientas adicionales, denominadas **plugins**, que facilitarán aún más la tarea de programación, depuración y verificación del código.

La captura de pantalla muestra la página web de Microsoft para las actualizaciones de Visual Studio Code. El encabezado indica "Version 1.26 is now available! Read about the new features and fixes from July.". La sección "UPDATES" muestra una lista cronológica de actualizaciones desde junio de 2017 hasta julio de 2018. La sección principal titulada "July 2018 (version 1.26)" incluye enlaces para descargas de Windows, Mac, Linux 64-bit y 32-bit, así como enlaces para "Workbench", "Languages", "Debugging", "Extensions", "Preview Features", "Extension Authoring", "Proposed Extension APIs", "Miscellaneous", "New Documentation" y "Notable Changes". Una flecha roja apunta a la lista de "Languages". A la derecha, una columna titulada "IN THIS UPDATE" enumera las mejoras mencionadas anteriormente. Abajo de la lista, hay enlaces para "Subscribe", "Ask questions", "Follow @code" y "Request features".

En code.visualstudio.com/updates encontraremos información sobre cada nueva release de este IDE.

Visual Studio Code

Desde hace algunos años, Microsoft cuenta con una herramienta totalmente gratuita llamada **Visual Studio Code**. Este IDE, desarrollado bajo el paradigma Open Source e inspirado en el editor de código **Atom**, está disponible para las plataformas Windows, Linux y Mac.

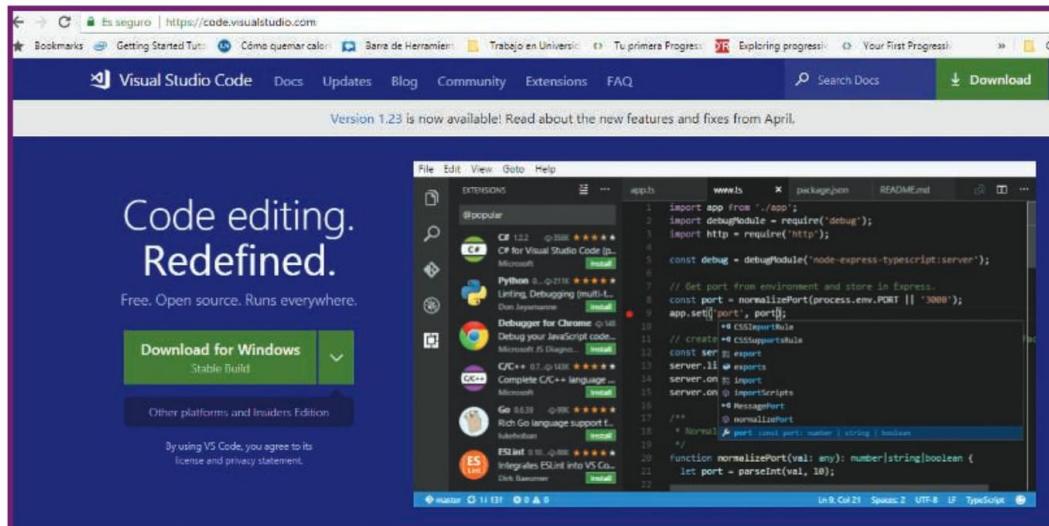
También existen otros editores con excelentes prestaciones para cualquier programador, por ejemplo Atom, Sublime Text o Notepad+.

Plugins esenciales

Existe un compendio de plugins subidos a la plataforma de Code que ofrecen diferentes opciones a los programadores, como ahorro de escritura de código mediante plantillas prearmadas, y acceso FTP para subir el contenido a una web, entre un sinnúmero de funciones más. Están accesibles desde el buscador de extensiones, presionando CTRL + SHIFT + X. Recomendamos siempre utilizar los plugins indispensables, ya que una instalación masiva de ellos ralentizará el IDE.

Configurar el IDE

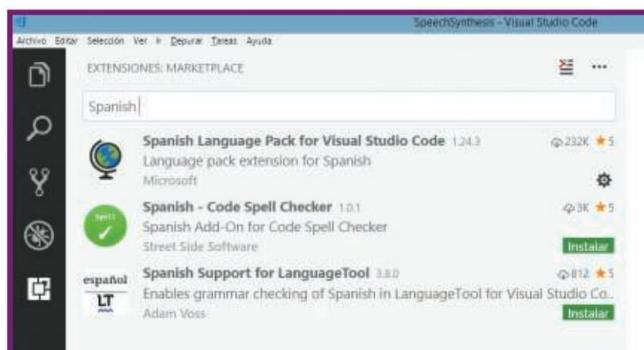
Visual Studio Code cuenta con un sistema de configuración que permite adaptar el IDE a nuestra necesidad. Pero, a diferencia del resto de las aplicaciones personalizables, en este IDE se modifica un archivo JSOn a nivel código.



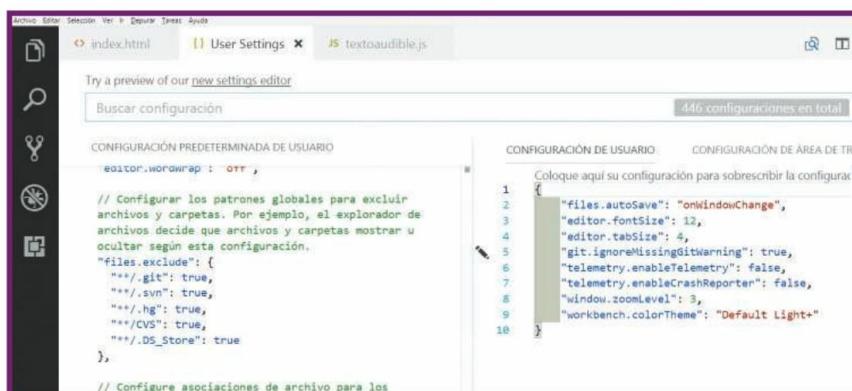
1 Ingresamos a la URL <https://code.visualstudio.com>, y allí encontraremos el link para descargar Visual Studio Code según nuestro sistema operativo actual. Realizado este paso, ejecutamos la instalación correspondiente.



2 El proceso de instalación es sumamente sencillo, guiado como siempre por una interfaz gráfica que nos explica y consulta sobre lo que vamos a instalar y debemos dejar configurado.



3 Ya dentro de Code, nos ocupamos de instalar los plugins **SpanishLanguage Pack**, que adaptan toda la interfaz a nuestra lengua madre; y **Live HTML Previewer**, que ejecuta una vista previa de forma integrada al IDE.



4 Presionamos la tecla CTRL seguida de la coma (,) para abrir la configuración del entorno. Ubicamos allí el parámetro que deseamos cambiar, y lo pegamos en el panel derecho modificando su valor por el deseado.

VS Code

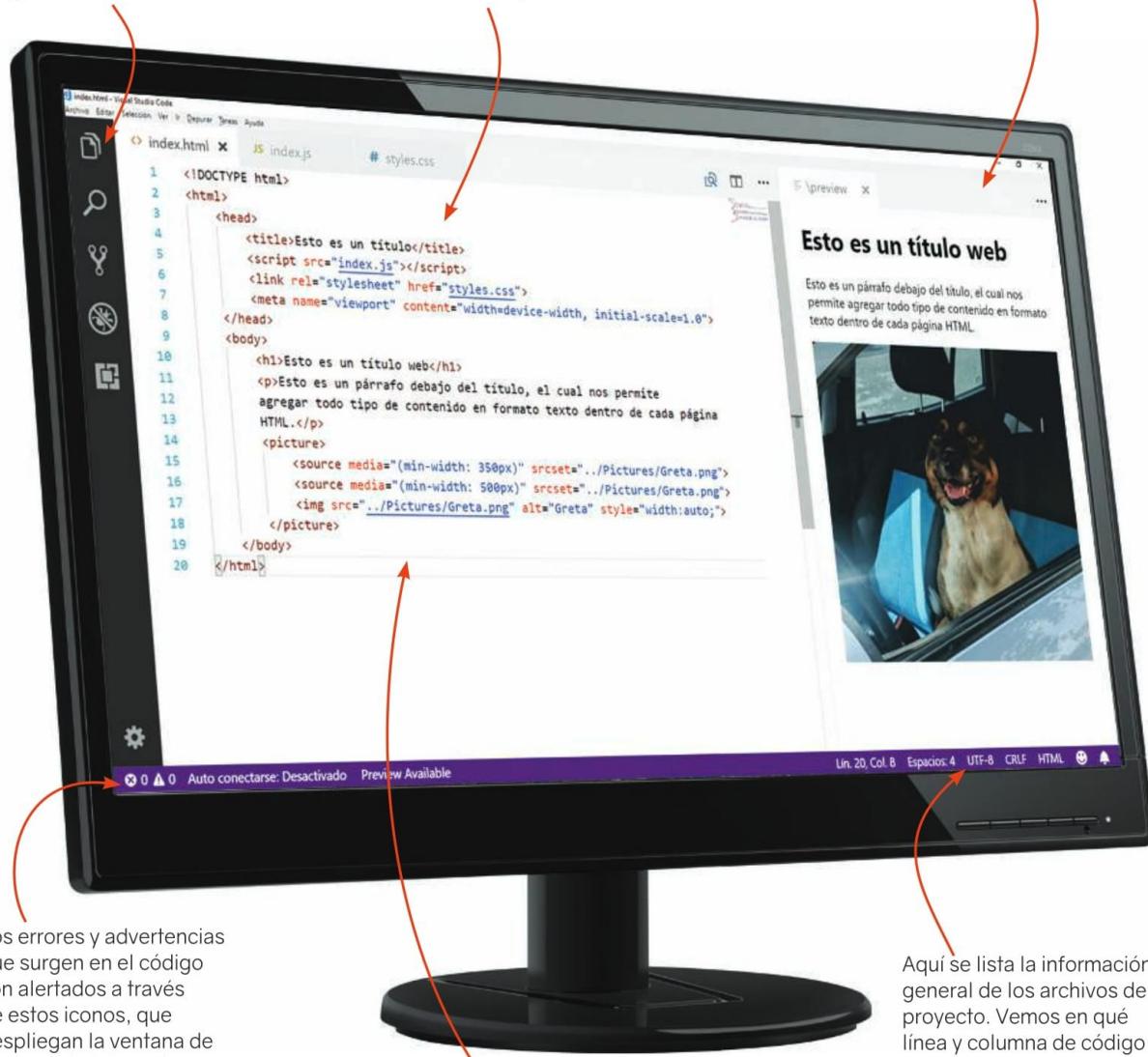


VS Code es un completo editor de código fuente que incluye decenas de funcionalidades (o plugins) desarrolladas por terceros, ideales para complementar nuestros proyectos.

A través de la barra lateral, accedemos rápidamente a la lista de archivos y carpetas de nuestro proyecto, buscamos textos o cadenas de caracteres específicas dentro de los archivos de trabajo, nos conectamos y sincronizamos el contenido contra Github, accedemos a la ventana de depuración de código, y sumamos más plugins dentro de VS Code.

Todos los archivos que se incluyen en nuestro proyecto (HTML, CSS, JS, Imágenes, JSON, etcétera) son abiertos y listados en esta barra superior. Con solo hacer clic sobre el archivo en cuestión, podremos ver su código y comenzar a trabajar sobre él.

Dentro de los complementos básicos que instalamos en Code, se encuentra Live HTML Previewer, el cual permite obtener una vista del contenido que estamos creando. Lo activamos presionando CTRL + Q + S.



Los errores y advertencias que surgen en el código son alertados a través de estos iconos, que despliegan la ventana de depuración integrada en Code y permiten obtener un listado de ellos. Haciendo clic en alguno, se nos conduce al archivo de código para corregirlo.

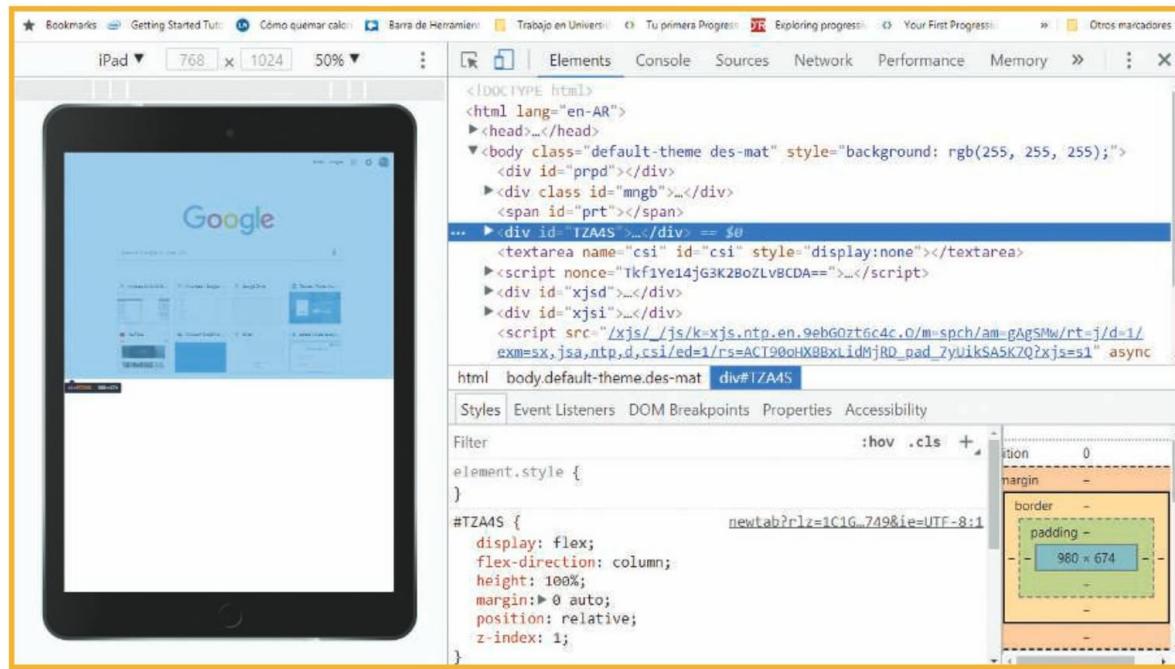
El código que editemos en cualquier archivo del proyecto aparecerá en esta área. Presionando CTRL+ / CTRL- podemos hacer zoom para tener una mejor visualización del código, según nuestra preferencia.

Aquí se lista la información general de los archivos del proyecto. Vemos en qué línea y columna de código nos encontramos, los espacios de tabulación, el código de idioma y el tipo de archivo, entre otras tantas advertencias y notificaciones propias de este IDE.

Herramientas para el desarrollador

Para depurar JavaScript a lo largo de esta obra, utilizaremos las Herramientas para el desarrollador. Estas vienen integradas a todos los navegadores web y nos permiten, entre otras opciones, depurar código; analizar variables; detectar errores; acceder al código fuente del frontend de cualquier sitio web; desplegar y contraer

bloques de código anidados HTML, CSS o JS; revisar el comportamiento de la red al cargar un sitio; verificar su performance y la memoria que consume; ver información sobre una web instalable, por ejemplo, Progressive Web App; y analizar el comportamiento responsivo.



Un mundo detrás de F12

Las **Herramientas para el desarrollador** o **Developer Tools** se activan en cualquier pestaña de navegación presionando la tecla F12. A lo largo de las próximas páginas, conoceremos más

en detalle sus funciones integradas. Si somos usuarios de otros navegadores web, aquí listamos las URL de las herramientas para el desarrollador de otros browsers muy populares:

NAVEGADOR WEB	AYUDA EN LÍNEA DE HERRAMIENTAS PARA EL DESARROLLADOR
Microsoft Edge	https://docs.microsoft.com/en-us/microsoft-edge/devtools-guide
Mozilla Firefox	https://developer.mozilla.org/en/docs/Tools
Google Chrome	https://developers.google.com/web/tools/chrome-devtools
Opera Browser	www.opera.com/dragonfly
Apple Safari	https://developer.apple.com/safari/tools

Herramientas para el desarrollador de los diferentes navegadores web.

Si bien utilizaremos Google Chrome como nuestra referencia para probar los desarrollos web en JavaScript, las herramientas antes mencionadas nos permitirán consultar y probar nuestros desarrollos en otras plataformas en caso de que lo necesitemos.

JS integrado versus JS independiente



En nuestro primer acercamiento a JavaScript, repasaremos las ventajas y desventajas de utilizar la sintaxis JavaScript integrada en un archivo HTML, versus la sintaxis JS en un archivo independiente. También desarrollaremos nuestro primer proyecto, el clásico "Hola Mundo", de todas las formas posibles que nos permite este lenguaje.

En los inicios de la era Web, JavaScript fue la estrella de fácil implementación dentro de cada documento HTML. La simplicidad de su código permitía llevar a las webs efectos notorios sobre imágenes y textos, e interactuar fácilmente con el browser. Esto fue un gran avance para la época, pero también fue un arma de doble filo, ya que las conexiones vía

módem hacían pesada la carga de contenido en abundancia. El resultado final fue que JS no beneficiaba a los websites de la primera era y, por consiguiente, su implementación fue estrictamente limitada y replanteada, de modo que no perjudicara la velocidad y la respuesta que el avance tecnológico de Internet había puesto ante nosotros.

The World Wide Web project

WORLD WIDE WEB

The WorldWideWeb (W3) is a wide-area hypermedia[1] information retrieval initiative aiming to give universal access to a large universe of documents.

Everything there is online about W3 is linked directly or indirectly to this document, including an executive summary[2] of the project, Mailing Lists[3], Policy[4], November's W3 news[5], Frequently Asked Questions[6].

What's out there?[7] Pointers to the world's online information, subjects[8], W3 servers[9], etc.

Help[10]	on the browser you are using
Software Products[11]	A list of W3 project components and their current state. (e.g. Line Modem[12], X11 Violin[13], NeXTStep[14], Servers[15], Tools[16], Mail robot[17], Library[18])
Technical[19]	Details of protocols, formats, program internals etc

{ref.number}, Back, {RETURN} for more, or Help:



En el link <https://bit.ly/1dV8npP> podemos experimentar cómo era una web y su velocidad de carga en la época del módem telefónico, sin CSS, ni imágenes... ¡ni JavaScript!

Si bien JavaScript era un lenguaje de programación independiente, en ese tiempo dependía de que los usuarios tuvieran instalada Java Virtual Machine en su

computadora. Con los años y el gran avance de los motores de navegación web, JavaScript se integró a ellos y facilitó su uso a la mayoría de los navegantes web.

Estructura HTML

Antes de sumergirnos en el mundo JS, veamos las bases del lenguaje de marcado HTML, el cual es fundamental para alojar todos los proyectos que realizaremos en esta publicación. Si ya tenemos el conocimiento necesario sobre esta tecnología, podemos pasar por alto esta sección, e ir directamente al desarrollo del primer proyecto.

En la actualidad, se utiliza HTML como lenguaje de desarrollo web. Este término proviene de **HyperText Markup Language**, o **lenguaje de marcado de hipertexto**. Este se encarga de indicarle al motor de navegación cómo debe formatear estructuralmente una web al momento de cargarla en la pestaña del navegador.

Veamos a continuación el código de ejemplo de un HTML base.

Código HTML base

<HTML>

Este tag hace referencia a cuándo comienza el contenido preformatado en el lenguaje de hipertexto. El contenido se inicia con el tag <html> y finaliza con el tag </html>. Todo lo que está entre ellos será analizado por el navegador web, que tratará de interpretarlo como tag HTML (una imagen, texto, un video, etcétera). En caso de que no pueda hacerlo, el motor de navegación mostrará el contenido en cuestión como un simple texto.

<!DOCTYPE HTML>

Es la declaración que todo documento HTML debe tener como primera instancia, precediendo al tag <html>. Esta instrucción le indica al navegador web qué versión de HTML se utilizó para escribir la página que está por cargar. Si bien HTML5 no requiere el uso de la declaración <!DOCTYPE...>, se la sigue empleando porque, muchas veces, en una página web se incluyen referencias de HTML moderno (HTML5) y referencias de HTML antiguo (HTML 4.1 o anterior).

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>Page Title</title>
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="stylesheet" type="text/css" media="screen" href="main.css" />
  </head>
  <body>
  </body>
</html>
```

<HEAD>

Se usa comúnmente para declarar el contenido principal de la página HTML, como título a mostrar, formato de caracteres utilizado, referencias a las hojas de estilo CSS, referencias a los archivos externos JavaScript, bloque de código JavaScript que se usa embedido en el HTML, icono de la web, y otros tantos recursos que complementan la página. Al igual que el resto de los tags que conforman HTML, este se cierra con la contraparte </HEAD>.

<BODY>

Dentro de este tag se agrupa todo el contenido que le da vida a la parte gráfica de la página HTML: títulos, textos, imágenes, videos y audios, entre otro material que aquí pueda volcarse. Todo esto finalmente conforma la página web en sí, que es lo que vemos como navegantes en nuestro browser. Todo tag declarado se cierra con el tag </BODY>, y cualquier otro tag gráfico que se declare fuera de él tal vez no sea interpretado por el navegador.

Esta infografía nos da un panorama general de lo que HTML necesita para funcionar. Ahondemos un poco más en los principales tags que se utilizan en los apartados <HEAD> y <BODY>.

Apartado HEAD

TAG	DESCRIPCIÓN
<meta charset="utf-8" />	El elemento <meta charset> se ocupa de indicarle al navegador la codificación de contenido utilizada en este DOC web. La referencia UTF-8 garantiza la codificación estándar Unicode. Es recomendable ubicar la etiqueta seguida a la apertura del tag <HEAD>.
<title>...</title>	Indica el título que lleva la página web en sí. Es lo que se visualiza en la pestaña del navegador cuando se carga un documento HTML.
<meta name="viewport">	A través de este elemento, podemos fijar el contenido del documento para que se ajuste al browser que lo carga (de escritorio, móvil, de tablet, etcétera).
<link rel="stylesheet">	Permite referenciar el archivo de estilos, comúnmente llamado hoja de estilos, hacia la ruta local o remota de un servidor web.

Apartado BODY

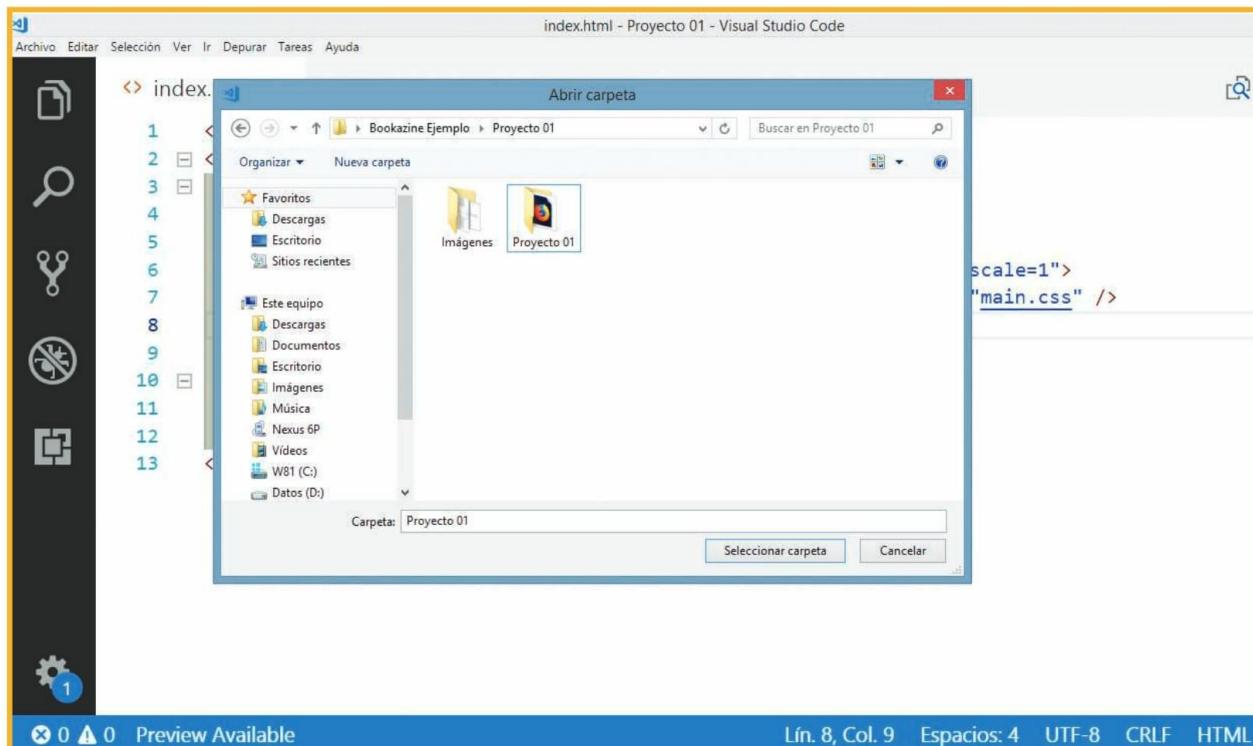
TAG	DESCRIPCIÓN
<div>...</div>	Permite encerrar en su interior la mayoría de los contenidos o etiquetas HTML que conforman la página web.
<h1>...<h6>	A través de H1, 2, 3, 4, 5 y 6 damos vida a los títulos que se incluyen antes de un texto extenso.
<p>...</p>	El elemento Paragraph nos permite incluir el texto que se utiliza dentro de cada sitio web.
	A través de referenciamos las imágenes que ilustran el documento HTML.
<video>	Este elemento nos permite insertar un video con los controles básicos (volumen, play, pausa, stop, pantalla completa, etcétera).
<audio>	Los archivos de audio en los formatos más comunes son visualizados junto con el set de controles básicos, a través de este tag HTML.



Si bien existe un sinfín de etiquetas HTML que conforman los documentos en cuestión, a través de estas tablas presentamos las más esenciales. Muchas de ellas serán usadas a lo largo de esta obra, y las crearemos, controlaremos y modificaremos desde el lenguaje JavaScript.

JS dentro del HTML

Comencemos a delinear nuestro primer trabajo, conocido como “**Hola mundo!**”. Para hacerlo, vamos a ingresar en Visual Studio Code y crear dentro de este IDE el primer proyecto, presionando la combinación de teclas **CTRL + K + O**.



Tal como vemos, Code nos pedirá abrir una carpeta.

También, en esta misma ventana, tenemos la opción de crear una nueva presionando el ícono correspondiente de la barra de herramientas superior. Hacemos esto, le damos un nombre a la carpeta, y acto seguido, presionamos **Abrir Carpeta**.

Con esto ya creamos nuestro primer entorno de proyecto. A continuación, presionamos **CTRL + N** para generar un nuevo archivo, que guardamos con el nombre **index.html**.

Ahora copiamos en este documento nuestro primer texto HTML, tal como muestra el siguiente bloque de código:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>Javascript - Proyecto 01</title>
  </head>
  <body>
    <h1>Nuestro primer código Javascript</h1>
  </body>
</html>
```

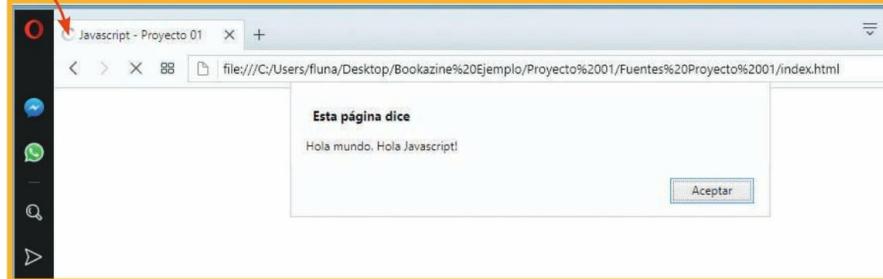
Como vemos hasta aquí, el código HTML es muy básico y prescinde de algunos elementos mencionados en las tablas de referencia. Esta base nos basta para desarrollar el primer ejemplo.

El tag <script>

A continuación, utilizaremos el tag **<script>**, que nos permitirá en esta instancia escribir el código JavaScript básico de nuestro primer ejemplo. El siguiente bloque de código se ubica antes del cierre de la etiqueta **</head>**:

```
<script>
    alert("Hola mundo. Hola Javascript!");
</script>
```

Guardamos el código escrito hasta el momento, y luego, arrastramos el archivo index.html hasta una nueva pestaña del navegador web. El resultado obtenido debe ser el que muestra la siguiente imagen:



Como podemos apreciar, hemos escrito nuestra primera sentencia JavaScript. En ella utilizamos la función **alert()** para visualizar un texto en pantalla.

Ahora probemos a desplazar el bloque de

código JS escrito hasta otra ubicación del archivo HTML. Vamos a ubicarlo precisamente entre el cierre del tag **</body>** y el cierre del tag **</html>**. El código quedará de la siguiente forma:

```
<!DOCTYPE html>
...
</body>
<script>
    alert("Hola mundo. Hola Javascript!");
</script>
</html>
```

Ejecutemos otra vez el archivo HTML, refrescando la página cargada en el navegador web mediante CTRL + R. ¿Logramos el mismo cometido que antes? ¿Pudimos volver a visualizar el mensaje escrito en JS?



Seguramente que sí, hemos conseguido lo mismo. Ahora, ¿vemos alguna diferencia entre una y otra prueba realizada?

El motor de un navegador web, cualquiera que este sea, busca la página web principal (o específica) en el servidor o URL que le indiquemos. Al encontrarla, la carga en la caché

local y, luego, la procesa leyendo tal como leemos un libro: de arriba hacia abajo. Analiza cada uno de los tags de la página, los procesa, y dibuja en la pestaña del navegador web, de manera progresiva, lo que el tag, código script u hoja de estilo CSS indica.

Comportamiento de HTML y JS en un web browser

A través de la siguiente infografía veamos cómo se comportan un navegador web y un sitio web durante el proceso de solicitud de una página, su devolución, procesamiento y visualización del código.

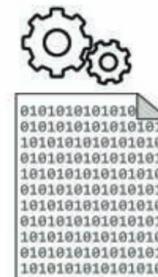
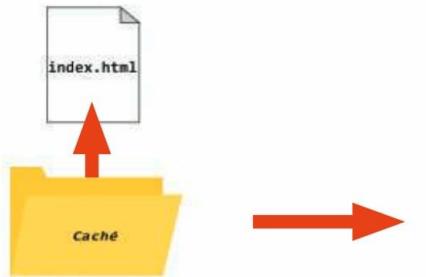
Petición



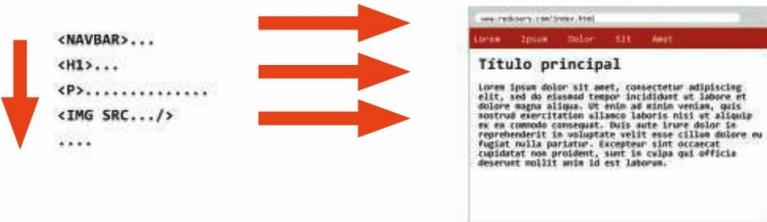
Cuando un navegador web hace la petición de una página o sitio a Internet, envía la URL requerida hacia la gran nube. Se producen todas las redirecciones necesarias hasta encontrar la dirección. El servidor procesa el requerimiento, ejecuta procesos internos y, finalmente, le devuelve al navegador la página solicitada, o le indica cuál es la página principal o alternativa a la buscada.

Devolución y caché

Llega la página solicitada por el web browser, y como primera medida, el navegador se ocupa de almacenarla en una caché de datos. Luego, de allí pasa directamente al motor de navegación, que la procesa para, finalmente, mostrar el contenido en pantalla.



Procesamiento y visualización



Por último, el motor del navegador web analiza y “decodifica” la sintaxis de la página, y va mostrando en pantalla, progresivamente, cada uno de los tags que esta incluye. Es por eso que la carga de cada página de sitios web que solemos visitar se ve de forma gradual.

Pero ¿por qué explicamos esto? Esta manera de procesar la información que tienen los navegadores web influye no solo en el código HTML y CSS de cada página, sino también en el código JavaScript que esta contenga, ya

sea embebido en el documento HTML o en un archivo aparte. Es por eso que, en determinadas situaciones, sí influirá el lugar donde ubicaremos el código JavaScript o la referencia a nuestro archivo JS.

JavaScript fuera del HTML



Veamos ahora cómo llevar adelante el mismo ejemplo, pero utilizando un archivo externo al documento HTML. Los archivos que alojan contenido exclusivo de JavaScript finalizan con la extensión .JS.

Volvamos al proyecto, eliminemos el código dentro del tag <script>, y creamos un nuevo archivo, que llamaremos index.js. Dentro de él, escribimos el siguiente código:

```
function saludo() {  
    alert("Hola mundo. Hola Javascript!");  
}
```

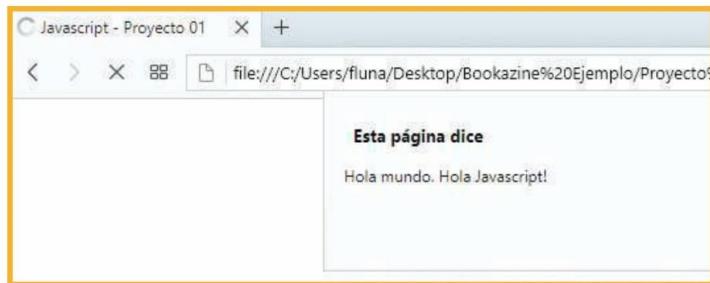
Luego, modificamos el archivo index.html, agregando dentro del tag <head>, lo siguiente:

```
...  
<script src="index.js"></script>  
</head>
```

Por último, modificamos el tag de apertura <body>:

```
<body onload="saludo();">
```

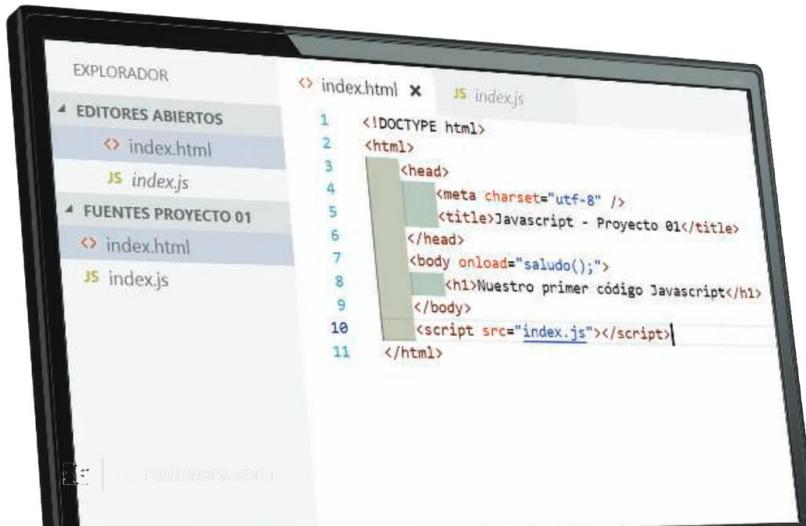
El resultado debe ser igual al que se muestra en la imagen siguiente.



Si ejecutamos el ejemplo con esta nueva modificación, veremos que llegamos al mismo camino que con el código escrito anteriormente, con la diferencia de que nuestro código JS está escrito en un archivo separado, y que el bloque de código que ejecuta el mensaje "**Hola mundo...**" está contenido dentro de una función.

Más adelante analizaremos para qué se usan las funciones.

Ahora, como última prueba, modifiquemos la ubicación de la línea <script> dentro del HTML. Ubiquémosla entre el cierre del tag </body> y el cierre del tag </html>. Por último, volvamos a probar nuestro ejemplo para ver si conseguimos el mismo resultado.



JavaScript mediante consola

Tomemos el último ejemplo que realizamos, y editemos el archivo index.js, agregando en él las siguientes líneas de código:

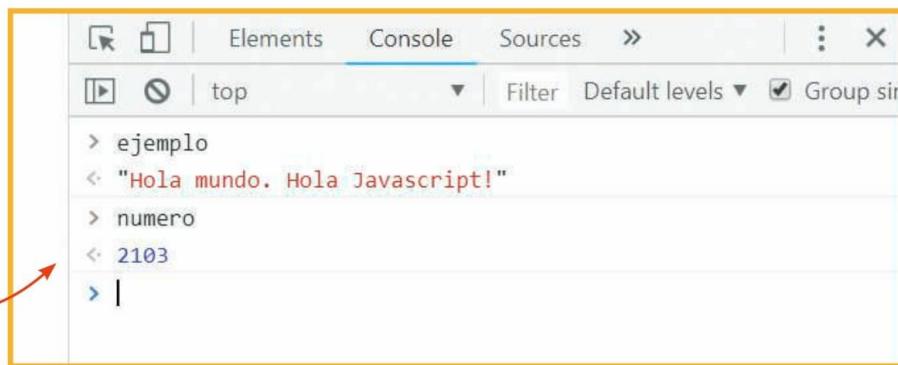
```
var ejemplo = "Hola mundo. Hola Javascript!";
var numero = 2103;
```

Guardamos lo realizado y abrimos el proyecto en Google Chrome. Seguido a esto, presionamos la tecla F12. Se abrirá en el mismo navegador web el complemento llamado Herramientas para el desarrollador.

Ubicamos la pestaña **Consola** y presionamos sobre ella, posicionando a continuación el cursor en la línea de comandos (la identificamos con el símbolo >). Allí escribimos primero la variable ejemplo seguida de ENTER.



Luego escribimos la variable numero también seguida de ENTER.



Como podemos apreciar en la consola de **Herramientas para el desarrollador**, esta nos permite consultar el contenido de las variables JS, asignar valores a variables, crear nuevas, ejecutar una función JS, y otras tantas opciones más. Las herramientas para el desarrollador de Google Chrome serán nuestras aliadas durante toda esta obra, por lo que recomendamos utilizar este navegador web para las pruebas principales de

cada uno de los proyectos que vamos a elaborar. También, como alternativa, podemos recurrir a **Chromium**, que cumple las mismas funciones que Google Chrome. Luego de nuestro primer acercamiento a JavaScript, ahora es el turno de conocer las principales características de este lenguaje, las cuales debemos tener en cuenta para programar los próximos proyectos.

Sintaxis básica de JavaScript



Veamos a continuación cuáles son las características más destacables del lenguaje JavaScript, y de qué manera podemos usar o crear funciones, así como también manejar objetos.

PALABRAS RESERVADAS

Toda palabra reservada es identificada por el lenguaje con otro color.

DECLARACIÓN DE VARIABLES

Las variables son declaradas anteponiendo la palabra var. El nombre de una variable debe ir en minúsculas y tiene que ser posible identificar su contenido a través de su nombre. No hay que usar caracteres especiales, ni tildes, diéresis, ni nada que complique su nombre.

TIPOS DE DATOS

A diferencia de otros lenguajes de programación, las variables no requieren definir un tipo de dato; pueden almacenar un texto, número o valor booleano, sin problema.

ASIGNACIÓN DE DATOS

Para asignar un dato a una variable, siempre es necesario anteponer el signo igual (simple).

TEXTOS

Cuando asignamos un texto a una variable, o dentro de una función, este debe estar encerrado entre comillas (dobles o simples).

FINAL DE UNA SENTENCIA

Cada línea de código que finaliza debe llevar punto y coma (;).

NÚMEROS Y BOOLEANOS

Al asignar un número a una variable o función, este debe estar escrito sin comillas. Lo mismo vale para los valores booleanos (True o False).

FUNCIONES PERSONALIZADAS

Cuando creamos una función en JS, esta lleva antepuesta la palabra reservada function, seguida del nombre.

FUNCIONES INTEGRADAS

Son parte del lenguaje y ya tienen predefinida su tarea, como también si reciben parámetros o no.

DECLARACIÓN DE PARÁMETROS

Al crear un nombre de función, hay que terminarlo con paréntesis. Los argumentos o parámetros se definen dentro de los paréntesis.

USO DE LLAVES

Cuando creamos funciones, todo lo que ellas contienen debe ir encerrado entre llaves {...}, que ofician como limitadoras del contenido de la función en sí.

```
function saludo() {  
    alert("Hola mundo. Hola Javascript!");  
}
```

FUNCIONES INTEGRADAS

Son parte del lenguaje y ya tienen predefinida su tarea, como también si reciben parámetros.

TRABAJO CON OBJETOS

Dado que JavaScript permite crear y manipular objetos, estos se declaran tan fácilmente como cuando se crea una simple variable.

```
var automovil = new Object();
```

```
automovil.marca = "Kia";  
automovil.modelo = "Picanto";  
automovil.año = 2009;
```

```
var vehiculo = automovil.marca + ' ' + automovil.modelo;
```

ACCEDER A PROPIEDADES

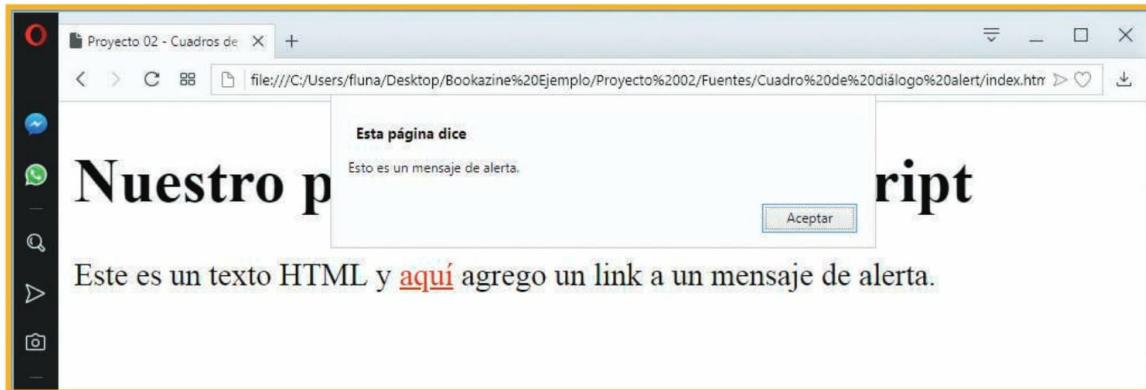
El consumo de las propiedades y métodos de un objeto es muy simple: **propiedad.objeto**, y así se podrá acceder al valor almacenado.

Cuadros de diálogo

Los cuadros de diálogo, tanto en JS como en cualquier otro lenguaje, son herramientas a través de las cuales brindamos o solicitamos información al usuario. Particularmente en JavaScript, existen tres tipos de cuadros de diálogo

predeterminados, que nos permiten insertar una interacción con el usuario. Estos son:

- **Alert()**
- **Confirm()**
- **prompt()**



Estos cuadros de diálogo son métodos que nos facilitan la interacción con el usuario y están alojados bajo el objeto Window. Veamos a continuación un detalle de cada uno de ellos.

alert()

El cuadro de diálogo **alert()** es el más simple de los tres. Se ocupa de mostrar en pantalla un mensaje personalizado por el desarrollador, y de integrar un simple botón, denominado **OK** o **ACEPTAR**.

La interacción del usuario con **alert()** es muy simple: consiste en leer el mensaje y aceptarlo una vez leído. Este cuadro funciona como un punto de interrupción en el código, lo cual indica que, hasta tanto el usuario no presione el botón Aceptar, el código que sigue no podrá procesarse.

Veamos un ejemplo de código:

```
alert("Este es un mensaje de texto.");
```

A través de esta línea, simplemente mostramos un mensaje de texto al usuario, aprovechando las capacidades de **alert()**. Ahora personalicemos un poco el mensaje que se va a mostrar, integrando una variable en él:

```
var mensaje = "Este es un mensaje  
para el usuario.";  
alert(mensaje);
```

Probemos directamente estos ejemplos en la consola de Herramientas para el desarrollador de nuestro navegador web. Allí veremos cómo se comporta el cuadro de diálogo.

```
> alert("Este es un mensaje de texto.");  
< undefined  
> var mensaje = "Este es un mensaje para el usuario.";  
< undefined  
> alert(mensaje);  
>
```

confirm()

Este otro cuadro actúa casi de la misma forma que el anterior, con la diferencia de que agrega al diálogo dos botones: ACEPTAR – CANCELAR. Como desarrolladores, nos permite capturar la opción que presionó el usuario y, en base a esta condición, ejecutar un bloque de código u otro. Veamos un ejemplo:

```
var respuesta = confirm("¿Está de acuerdo con nuestros Términos y Condiciones?");
```

En este caso, declaramos una variable denominada respuesta, y a ella le asignamos el resultado del cuadro de diálogo **confirm()**. Dado que este cuadro presenta dos opciones como respuesta, **ACEPTAR** o **CANCELAR**, el valor que almacenará esta variable será un booleano (**verdadero-falso**).

```
< undefined
> var respuesta = confirm("¿Está de acuerdo con nuestros
Términos y Condiciones?");
>
```

Si se presiona ACEPTAR, a la variable se le asigna el valor **TRUE**; mientras que si se presiona CANCELAR, se le otorga el valor **FALSE**. Escribamos el ejemplo del código anterior en la consola de las Herramientas para el desarrollador, y presionemos cualquiera de las opciones que nos da el cuadro de diálogo. Para ver qué valor queda asignado a la variable respuesta, vamos a escribir lo siguiente:

```
console.log(respuesta);
```

```
> console.log()
< undefined
> var respuesta = confirm("¿Está de acuerdo con nuestros
Términos y Condiciones?");
< undefined
> console.log(respuesta)
true
< undefined
> | VM201:1
```

Este tipo de cuadro de diálogo puede combinarse con un condicional, como la sentencia **If**, para así controlar de mejor manera el comportamiento de un algoritmo.

El objeto console

El lenguaje JavaScript cuenta con el objeto **console**, que permite acceder o ejecutar determinados mensajes en la consola del navegador o de nuestro IDE de desarrollo. El uso más común de **console** es el que vimos anteriormente, aunque también cuenta con opciones como **console.info()** y **console.warn()** entre otras. Las veremos en detalle más adelante.

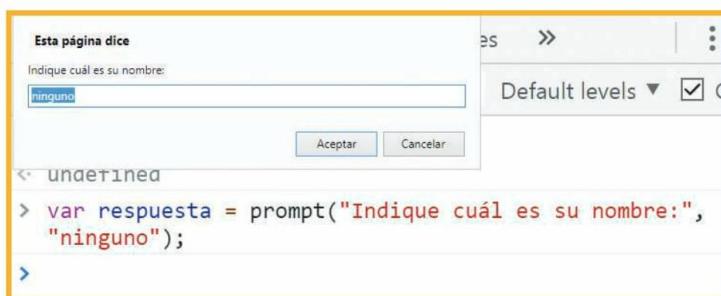
Prompt()

El método **prompt()** es un cuadro de diálogo de entrada que da la posibilidad de que el usuario ingrese un valor de forma manual, para que luego el algoritmo capture dicho input, lo procese, guarde, y/o muestre en pantalla.

Veamos cómo se utiliza este método:

```
var respuesta = prompt("Indique cuál es su nombre:", "ninguno");
```

Esta línea de código presenta en pantalla el cuadro de diálogo **Prompt()**, tal como lo muestra la siguiente imagen:



Veamos cómo se conforma este método, a través de la siguiente tabla:

SENTENCIA	DESCRIPCIÓN
var respuesta	Declaramos la variable en cuestión, la cual almacenará el resultado de la acción sobre el prompt.
prompt(param1, param2);	El método Prompt() tiene dos parámetros que debemos pasar. El primero corresponde al mensaje que verá el usuario en el cuadro de diálogo. El segundo es un valor predeterminado; si no queremos poner nada, simplemente dejamos doble comilla.
Botones ACEPTAR CANCELAR	Al igual que en el método confirm(), hay dos opciones a través de los botones mencionados. Si presionamos CANCELAR, la variable recibirá un Null como resultado, mientras que si pulsamos ACEPTAR, la variable almacenará el valor ingresado en la caja de texto, o el parámetro predeterminado.

Copiamos el bloque de código anterior a la consola de Google Chrome. Vamos a ejecutarlo y, cuando aparece el cuadro de diálogo **Prompt()**, ingresamos nuestro nombre. Luego, presionemos el botón ACEPTAR. En la consola escribamos:

```
console.log(respuesta);
```

Tendremos que ver en pantalla el nombre que ingresamos a través de esta caja de diálogo, tal como se observa en la imagen.

```
> var respuesta = prompt("Indique cuál es su nombre:", "ninguno");
<- undefined
> console.log(respuesta)
Fernando Luna
<- undefined
>
```

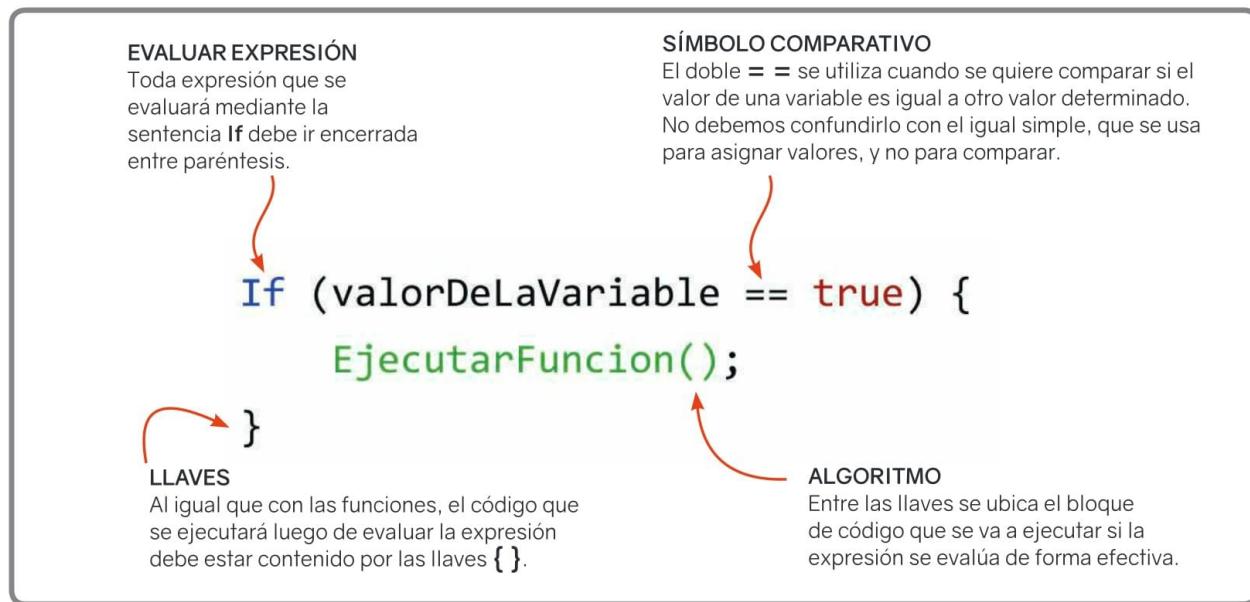
Condicionales: If

Un condicional es una instrucción que evalúa si una sentencia es verdadera o falsa y, sobre la base del resultado de dicha evaluación, ejecuta o no cierto código.

La sentencia **If** es un ejemplo de un condicional, que suele usarse para analizar una variable, operación o propiedad de un objeto. Como resultado, siempre obtendremos un valor booleano (**TRUE** o **FALSE**).

```
If (valorDeLaVariable == true) {
    EjecutarFuncion();
}
```

Veamos en la infografía las características básicas de esta sentencia.



En la sentencia **If**, la evaluación de una expresión no está condicionada solo por el símbolo de igualdad; también se pueden realizar comparaciones mayores, menores, distintas y hasta combinadas, para obtener un resultado determinado.

OPERADOR	DESCRIPCIÓN	EJEMPLO
<code>==</code>	Verifica una igualdad entre dos elementos, y retorna verdadero si la hay.	<code>If (valorDeLaVariable == true)</code>
<code>!=</code>	Verifica la no igualdad, retornando verdadero si es que no hay igualdad.	<code>If (valorDeLaVariable != 20)</code>
<code>></code>	Valida si el valor de la variable es mayor que el del resultado.	<code>If (valorDeLaVariable > 25)</code>
<code><</code>	Valida si el valor de la variable es menor que el del resultado.	<code>If (valorDeLaVariable < 30)</code>
<code>>=</code>	Valida si el valor de la variable es mayor o igual.	<code>If (valorDeLaVariable >= 35)</code>
<code><=</code>	Valida si el valor de la variable es menor o igual.	<code>If (valorDeLaVariable <= 40)</code>

Alternando Condicional: else

Como vimos, la sentencia **If** permite ejecutar un bloque de código siempre que la expresión evaluada dé como resultado True. Ahora, si necesitamos ejecutar otro bloque de código si la expresión evaluada dio como resultado False, podemos integrar la sentencia **Else**. Veamos un ejemplo:

```
If (variableComparado == True) {  
    DioComoResultadoVerdadero();  
} else {  
    DioComoResultadoFalso();  
}
```

Como podemos ver, la sentencia **else** se utiliza inmediatamente después de la llave de cierre de **If**. Al igual que esta última, **else** necesita encerrar el bloque de código a ejecutar, dentro de llaves **{ }**. Veamos un ejemplo práctico.

Abrimos nuestro navegador web y vamos a las Herramientas para el desarrollador. Allí ubicamos la consola y escribimos lo siguiente:

```
var r = prompt('Ingrese su nombre: ', ''');
```

Presionamos ENTER para ver el cuadro de diálogo en cuestión. En este cuadro tenemos dos opciones: escribir nuestro nombre o dejarlo en blanco. Realicemos alguna de ellas al azar, y presionemos ACEPTAR.

Nuevamente en la consola, escribimos la siguiente línea de código y pulsamos ENTER al finalizar:

```
if (r != '') {alert('Bienvenido ' + r);} else {alert('Por favor, debe  
ingresar un texto válido.')}  
↳
```

```
< undefined  
> if (r != '') {alert('Bienvenido ' + r);} else {alert('Por favor,  
debe ingresar un texto válido.')}  
< undefined  
> var r = prompt('Ingrese su nombre: ', '')  
< undefined  
> if (r != '') {alert('Bienvenido ' + r);} else {alert('Por favor,  
debe ingresar un texto válido.')}  
>
```

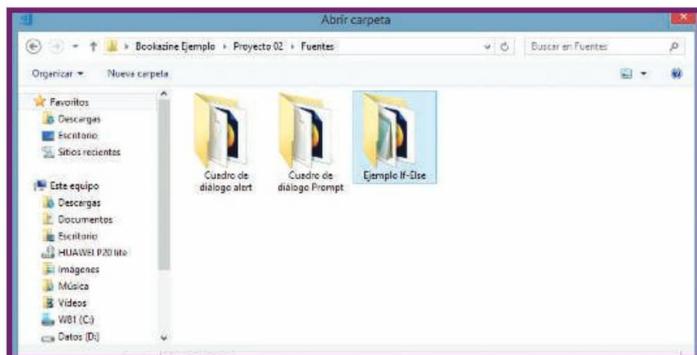
Como vemos en esta última línea de código, utilizamos **If** para validar si la variable **r** no está vacía y así saludar al usuario. Y con la sentencia **else**, cuidamos que el usuario ingrese un texto válido.

/<Ejercicio práctico>

Manejo de condicionales IF-ELSE



Llevemos a un ejercicio práctico las últimas líneas de código vistas. Realizaremos a continuación el manejo de IF-ELSE en conjunto con los cuadros de diálogo Prompt() y alert().



1 Creamos una carpeta para un nuevo proyecto con un nombre descriptivo, como **Ejemplo IF-ELSE**. A continuación, generamos un archivo **index.html** base, al cual le vinculamos en el apartado **<head>** un archivo denominado **index.js**.

```
index.html
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta charset="utf-8" />
5     <title>Proyecto 02 - Cuadros de diálogo Prompt</title>
6     <script src="index.js"></script>
7   </head>
8   <body onload="saludo()">
9     <h1>Prácticas IF-ELSE</h1>
10    <p id="textoSaludo"></p>
11  </body>
12 </html>
```

2 En el archivo **index.html** incluimos en el apartado **<body>**, un tag del tipo **título**, con el texto **Prácticas IF-ELSE**, y a continuación de él, un tag del tipo **<p>**, al que le asignamos un ID denominado “**textoSaludo**”.

```
index.js
1 saludo() {
2   resultado = prompt('Ingrese su nombre: ', '');
3   if (resultado != '') {
4     document.getElementById('textoSaludo').innerHTML = 'Bienvenido';
5   } else {
6     document.getElementById('textoSaludo').innerHTML = '';
7     alert('Por favor, ingrese un nombre.');
8 }
```

3 Luego de crear el archivo **index.html**, vamos a ocuparnos de **index.js**. En él creamos en primera instancia una función JS, llamada **saludo()**, la cual contendrá el cuadro de diálogo **Prompt**, junto con las sentencias **IF** y **ELSE**.



4 Volvemos a **index.html** y modificamos el elemento **<body>**, para que en el evento **onload()** de la página HTML se ejecute el llamado a la función **saludo()**. Guardamos todo y ejecutamos la página para probar nuestro proyecto.

Bloque de código JavaScript

```
function saludo() {  
    var resultado = prompt('Ingrese su nombre: ', '' );  
    if (resultado != '') {  
        document.getElementById('textoSaludo').innerHTML = 'Bienvenido ' + resultado;  
    } else {  
        document.getElementById('textoSaludo').innerHTML = '' ;  
        alert('Por favor, ingrese un nombre.' );  
    }  
}
```

Explicación del bloque JS

Lo primero que hacemos dentro de la función **saludo()** es declarar la variable **r**, la cual recibe de forma directa el valor ingresado (o no) a través del cuadro de diálogo **Prompt()**.

Luego, por medio de la sentencia **IF**, validamos que el valor de la variable **r** no se encuentre vacío. Esto lo hacemos a través de la comparativa **r != ''** (la variable **r** es distinta de vacío).

Si **r** tiene un valor asignado, entonces procesa

el bloque de código contenido en **IF**, el cual escribe el texto **Bienvenido + el nombre ingresado** en la página HTML.

Si a la variable no se le asignó ningún valor a través de **Prompt()**, entonces se ejecuta el bloque de código contenido en la sentencia **ELSE**. Este intercepta dentro del documento HTML un elemento cuyo **ID** es igual a **"textoSaludo"**, y elimina cualquier texto que se haya escrito.

Bloque de código HTML

```
<!DOCTYPE html>  
<html>  
  <head>  
    <meta charset="utf-8" />  
    <title>Proyecto 02 - Cuadros de diálogo Prompt</title>  
    <script src="index.js"></script>  
  </head>  
  <body onload="saludo()">  
    <h1>Prácticas IF-ELSE</h1>  
    <p id="textoSaludo"></p>  
  </body>  
</html>
```

Introducción a la Programación Orientada a Objetos

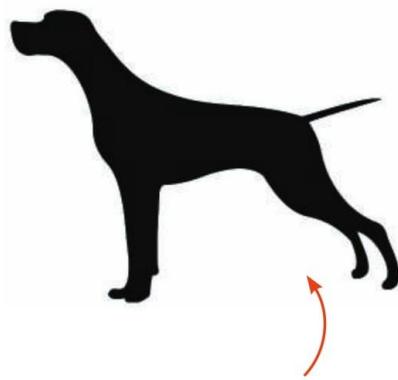


Una de las aplicaciones más comunes en JavaScript, al momento de programar, es la invocación de objetos que simplifican mucho las tareas cotidianas al escribir código. Veamos a continuación las principales características de estos elementos.

Dentro del mundo de la programación, el concepto “objetos” hace referencia a algo similar a lo que sucede en el mundo que nos rodea. Si miramos a nuestro alrededor, podemos ver un sinfín de cosas, como mesas, sillas, personas, puertas, ventanas, etcétera. A su vez, cada uno de estos objetos tiene un comportamiento

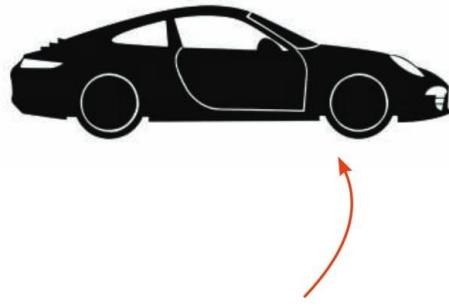
determinado; por lo tanto, si sabemos qué son y qué hace cada uno de ellos, entonces tendremos en claro su concepto.

Si tomamos como referencia un perro, sabemos que tiene cuatro patas, una cola y ladra. Si pensamos en un gato, también tiene cuatro patas y una cola, pero maúlla en vez de ladrar.



Características del Objeto Perro:

- 4 patas
- 1 cola
- 1 cabeza
- 2 ojos
- 1 boca
- ladra



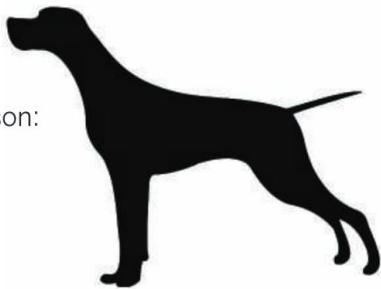
Características del Objeto Automóvil:

- 4 ruedas
- 2 puertas
- 1 baúl
- 1 volante
- 1 motor
- 2 asientos
- 2 faroles delanteros
- 2 faroles traseros
- 3 espejos

Por lo tanto, sin importar de qué tipo sean, cuando pensamos en objetos, podemos identificar que tienen cosas en común entre sí: los atributos y el comportamiento.

Atributos

Un atributo es una característica propia de un objeto. Si volvemos al ejemplo del perro, podemos identificar que algunos de sus atributos son:

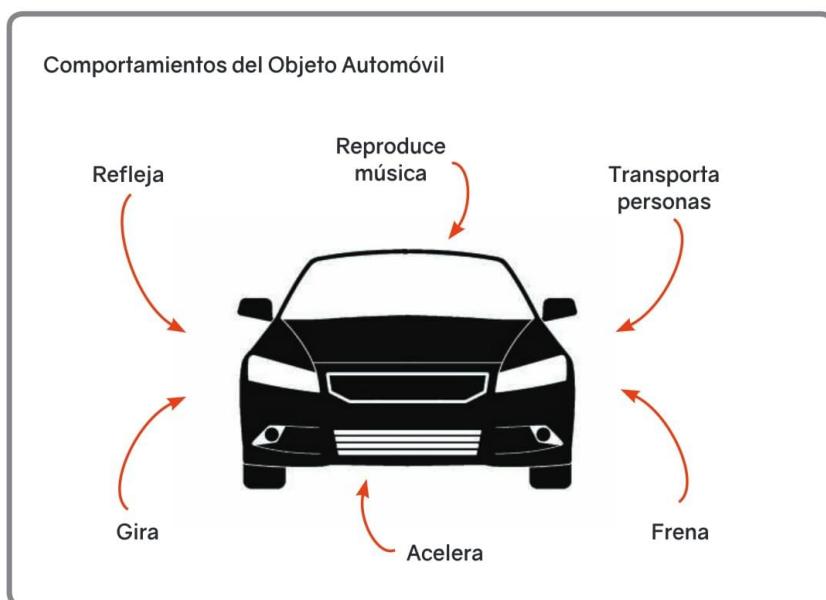


- altura
- color
- raza
- peso

Comportamiento

Pensando en el objeto automóvil, podemos decir que este tiene, entre otros, los siguientes comportamientos:

- acelerar
- girar
- frenar



Modelos de objetos

La programación orientada a objetos nos permite plasmar, en el mundo del código, modelos de objetos propios del mundo real. Como seres humanos, aprovechamos la capacidad de pensar, analizar y razonar, para poder diseñar de forma intuitiva, dentro del mundo de la programación, cada objeto que necesitamos utilizar, junto con sus principales características que lo distinguen del resto.

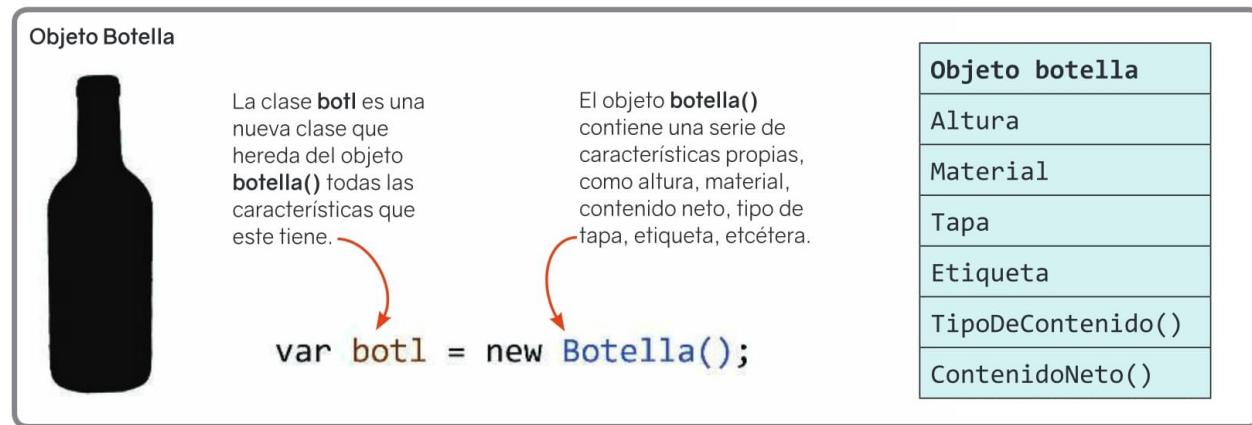
Los principales conceptos que nos brinda la programación orientada a objetos son:

- clases
- objetos
- atributos
- métodos
- mensajes



Clases

En JavaScript, cuando necesitamos trabajar con un objeto determinado, creamos una clase de él, que heredará sus atributos, métodos y propiedades. Veamos en la siguiente imagen un ejemplo de esto.



Cuando definimos la clase **botl**, hacemos que esta sea del tipo **Botella()**, con lo cual heredará todas las características de esta. Una vez creada la clase, podemos asignarle diferentes valores a cada uno de sus atributos, tal como lo representamos en la siguiente imagen:



Atributos

Podemos definir los atributos como diferentes variables que representan las características de un objeto. Cada atributo es un componente fundamental de un objeto, a tal punto que define su estado.

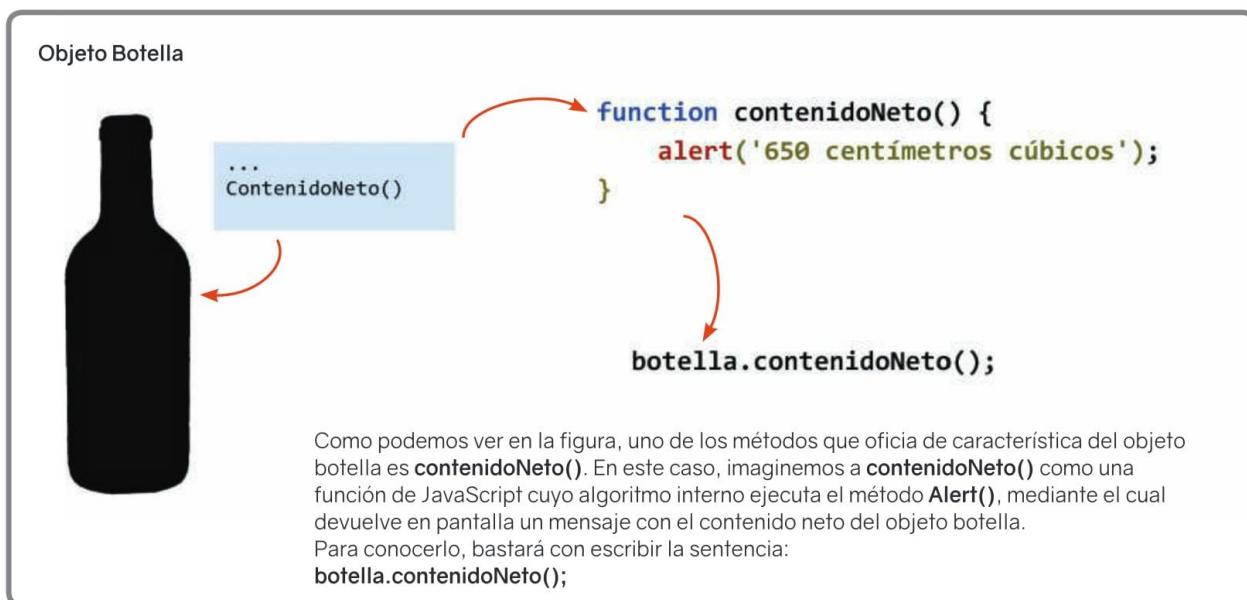


Mensajes

Un mensaje es una petición de un objeto a otro, para que este último realice una acción específica ejecutando uno de sus métodos. Los mensajes son los que se ocupan de relacionar entre sí los objetos que componen un programa.

Métodos

Son un conjunto de operaciones que un objeto puede realizar de acuerdo con su estado actual. Cada método que un objeto contiene son funciones que especifican una acción, de tal manera que definirán su comportamiento.

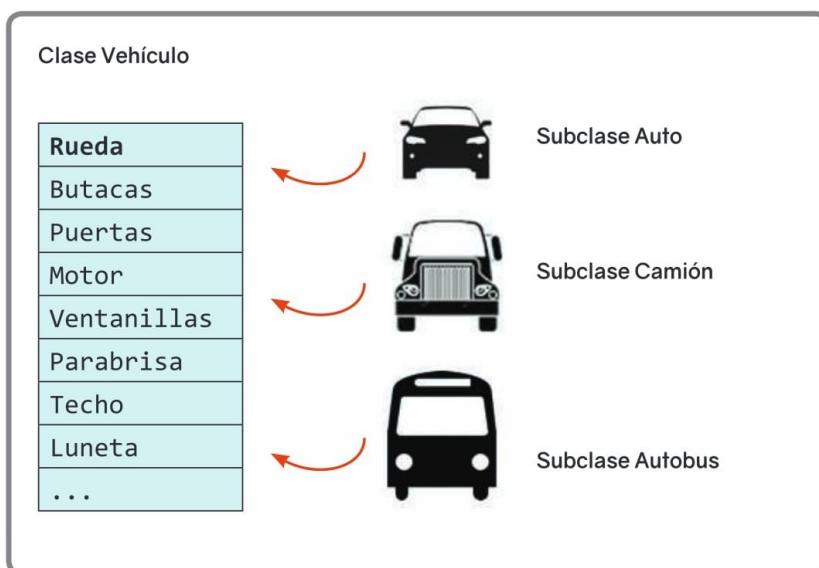


Herencia

La herencia nos permite crear una clase partiendo de un objeto determinado. Luego de hacerlo, podemos, a su vez, crear subclases de la clase principal, que heredarán todas o parte de las propiedades de aquella.

Para esto, imaginemos que tenemos una clase

Vehículo, de la cual crearemos subclases como: **Autos**, **Camiones**, **Autobuses**. Si la clase está definida con propiedades como **Ruedas**, **Butacas**, **Puertas**, **Motor**, **Ventanillas**, **Parabrisa**, **Techo**, **Luneta**, etcétera, las subclases heredarán todos estos atributos.



Características de un desarrollo orientado a objetos

Para que un desarrollo de software cumpla con el paradigma de orientación a objetos, debe tener las siguientes características:

- Herencia
- Encapsulamiento
- Abstracción
- Polimorfismo

Encapsulamiento

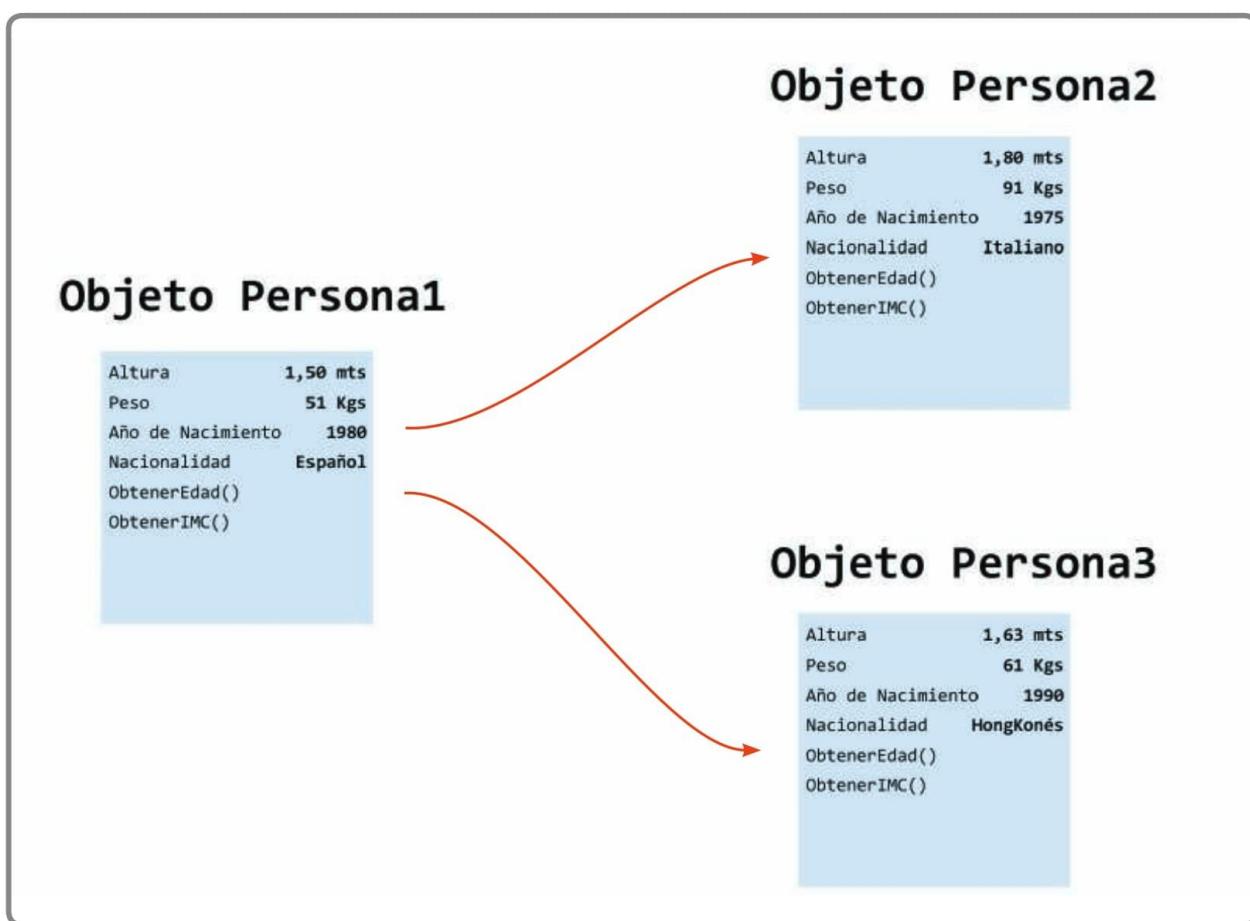
La información de un objeto es propia de él, y para acceder a ella, debemos hacerlo mediante un método predefinido, y no de forma directa, leyendo el atributo.

Abstracción

La abstracción hace referencia a la extracción o acceso de las propiedades esenciales de un concepto. Nos ayuda a no enfocarnos en los detalles que no son útiles o importantes respecto de lo que estamos realizando.

Polimorfismo

El polimorfismo nos garantiza que una operación determinada pueda tener diferentes comportamientos en distintos objetos, pudiendo estos reaccionar de manera determinada en el momento de enviar o responder una petición.



Tomando como premisa la imagen anterior, vemos un mismo objeto “persona”, con los mismos atributos y métodos pero diferentes valores en cada uno de ellos. Por ejemplo, si enviamos el mensaje **obtenerEdad()** a **persona1**, nos devolverá como resultado **38** (tomando 2018 como el año actual), mientras

que mandar el mismo mensaje a **persona2** y **persona3** nos devolverá como resultado **43** y **28** respectivamente. Así funciona el concepto de polimorfismo, aplicando la premisa de que “un mismo mensaje a distintos objetos puede devolver diferentes resultados”.

Fechas, intervalos y cronómetros



A través de este proyecto, conoceremos las herramientas que incluye JavaScript para manejar intervalos de tiempo, como así también para realizar cálculos temporales, desde nuestro código.

En la siguiente infografía veremos cuáles son las características que JS pone a nuestra disposición para controlar intervalos y espacios de tiempo en nuestros desarrollos de software.

SetInterval

setInterval(funcion, intervalo);

setInterval
Es un método de JS que se utiliza para invocar una función o evaluar una expresión determinada, durante un intervalo de tiempo especificado en milisegundos.

funcion
Es el parámetro de función o expresión que deseamos que se evalúe a través del método **setInterval**. Si es una función, puede recibir parámetros adicionales, en caso de requerirlos.

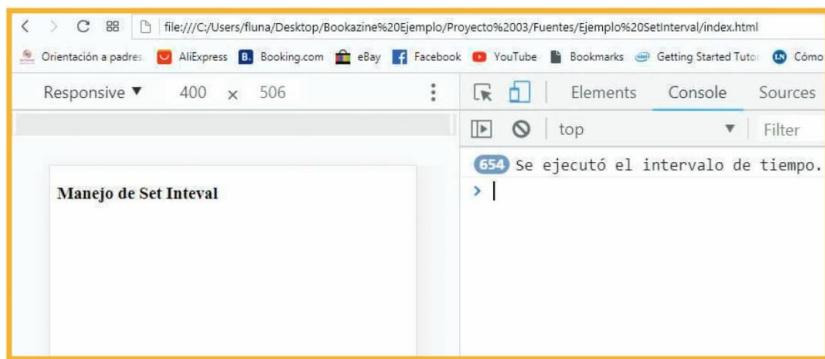
intervalo
Es el tiempo que debe pasar entre la ejecución de la función indicada. Se expresa en milisegundos (1 segundo dividido 1000), y el intervalo mínimo que podemos expresar es 10 ms.

Veamos a continuación un ejemplo de cómo funciona. Crearemos una página HTML base y un archivo JS denominado **index.js**, dentro del cual escribimos el siguiente código:

```
function intervalo() {  
    console.log('Se ejecutó el intervalo de tiempo.');//  
}  
var mv = setInterval(intervalo, 4000);
```

¿Qué hace este código? Crea la función llamada **intervalo**, la cual escribe una leyenda mediante `console.log` en la consola de depuración del navegador. Luego llama al método **setInterval()**, que ejecuta la función cada 4 segundos.

A continuación, nos aseguramos de que en el archivo **index.js** esté referenciado nuestro HTML, y ejecutamos el proyecto en el navegador web, desplegando las herramientas para el desarrollador. Allí visualizamos, cada cuatro segundos, que se ejecuta la leyenda indicada, tal como vemos en la imagen:



SetTimeout

`setTimeout(() => {funcion}, 9000);`

setTimeout

Es el método de JS utilizado para ejecutar una función o evaluar una expresión, luego de un determinado periodo de tiempo.

void

Impide que la función o método utilizado pueda invocarse a si mismo.

funcion

Es el parámetro de función o expresión que deseamos que se evalúe a través del método en cuestión. Si es una función, puede recibir parámetros adicionales, en caso de requerirlos.

intervalo

Es el tiempo que debe pasar entre la ejecución de la función indicada. Se expresa en milisegundos (1 segundo dividido 1000), y el intervalo mínimo que podemos indicar es 0 ms.

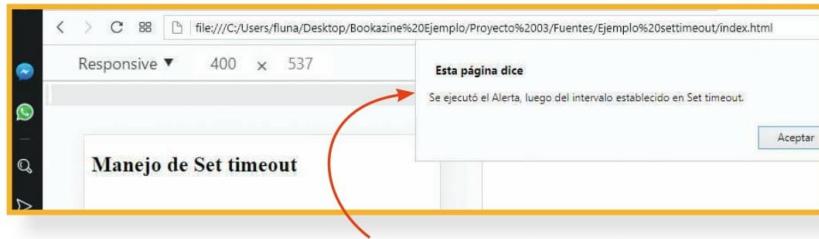
Los casos más habituales en los que podemos aplicar `setTimeout` son, por ejemplo, cuando llamamos datos remotos y estos llegan de forma asíncrona. Con esta función podremos priorizar o retrasar la ejecución de un bloque de código o de una función, luego de un período de tiempo estipulado.

Ejemplo de SetTimeout

Modifiquemos el código del ejemplo anterior, creando en el archivo JS la siguiente función:

```
setTimeout(() => {
    alert('Se ejecutó el Alerta, luego del intervalo establecido en Set timeout.')
}, 9000);
```

Como podemos ver en el código, establecemos que, luego de un período de 9 segundos, se ejecute la función `Alert()` en la aplicación. Ejecutemos el código en el navegador web para observar su comportamiento.



Como podemos apreciar, a diferencia de `setInterval()`, el método `setTimeout()` se ejecutará solo una vez.

Detener la ejecución de ambos métodos

Existen métodos que nos permiten detener la ejecución preestablecida de `setInterval()` y `setTimeout()`. Veamos a continuación cómo utilizarlos:

`clearTimeout(variable);`

clearTimeout()
Permite detener la ejecución del método `timeout()`, si este fue iniciado previamente.

variable: Dado que podemos iniciar múltiples métodos `setTimeout()` dentro de nuestra aplicación, necesitaremos crear cada uno de ellos en una variable determinada: `var to1 = setTimeout(...);`. Esto nos permitirá invocar luego el método `clearTimeout()`, especificando la variable `to1`, como parámetro del método, para así poder detener la ejecución. Si invocamos `clearTimeout()` sin parámetro adicional, se detendrán todos los métodos `setTimeout()` creados en el programa.

`clearInterval(variable);`

clearInterval()
Este método permite detener una función `setInterval` previamente inicializada.
variable: Al igual que lo explicado en `clearTimeout()`, debemos inicializar previamente el método `setInterval()` en una variable, para luego poder detener un método específico, en caso de que creamos más de uno en el programa.

El objeto Date()

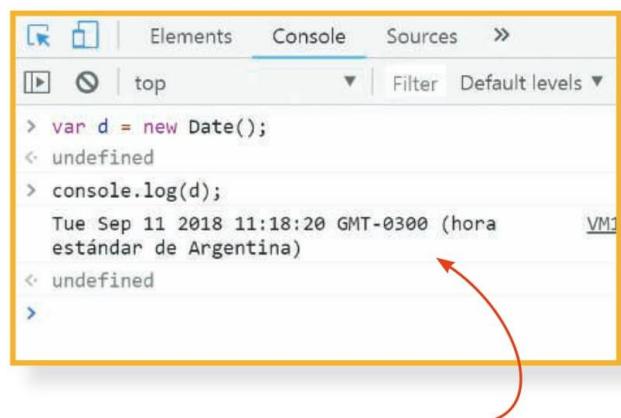
Habiendo definido los fundamentos de la programación orientada a objetos, nos ocuparemos de conocer el objeto **Date()**, fundamental para numerosos procesos y operaciones con JS que involucran fechas y horas,

```
//Creación de un objeto tipo Date  
Var d = new Date();
```

Probemos a escribir esta sentencia directamente en la consola de las Herramientas para el desarrollador de nuestro navegador web. Una vez escrita, ejecutemos:

```
console.log(d);
```

ya que simplifica su manejo. Se invoca a través del constructor **new Date()**, y permite obtener datos sobre un período de tiempo que incluyen **año, mes, día, horas, minutos, segundos y milésimas**, o especificar unidades entre todos ellos.



Como podemos apreciar en la imagen, a través de la consola del navegador, recibimos la información completa de la fecha y hora actuales. Tengamos presente que cada navegador web procesa de manera automática la fecha y la hora, propias del equipo del usuario.

Personalizar fechas

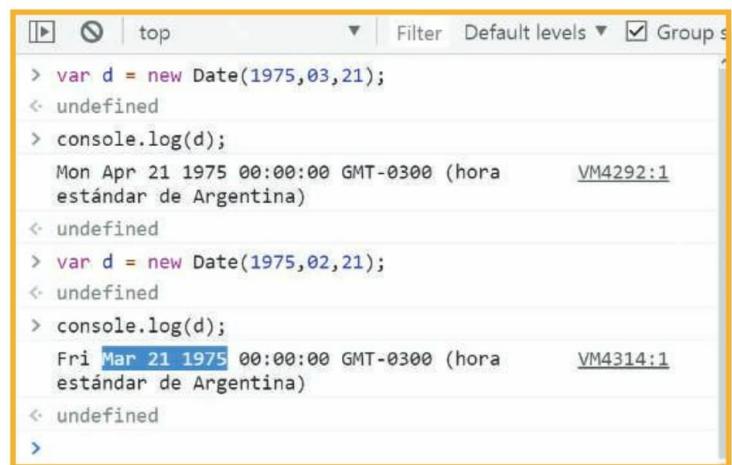
A través del objeto **Date()**, no solo podemos obtener la información de un día u hora específica, sino que también podemos crear un objeto del tipo **Date**, con una fecha especificada por nosotros:

```
Var d = new Date(1975, 02, 21);
```

Luego, mediante `console.log`, obtendremos el siguiente resultado:

```
console.log(d);  
VM4314:1  
Fri Mar 21 1975 00:00:00 GMT-0300 (hora estándar de Argentina)
```

Ahora, ¿existe alguna diferencia en la respuesta que devolvió `console.log`? Un dato: hay que prestar atención al mes. JavaScript contabiliza los meses desde el 0 hasta el 11, siendo enero el número 0, y diciembre el número 11.



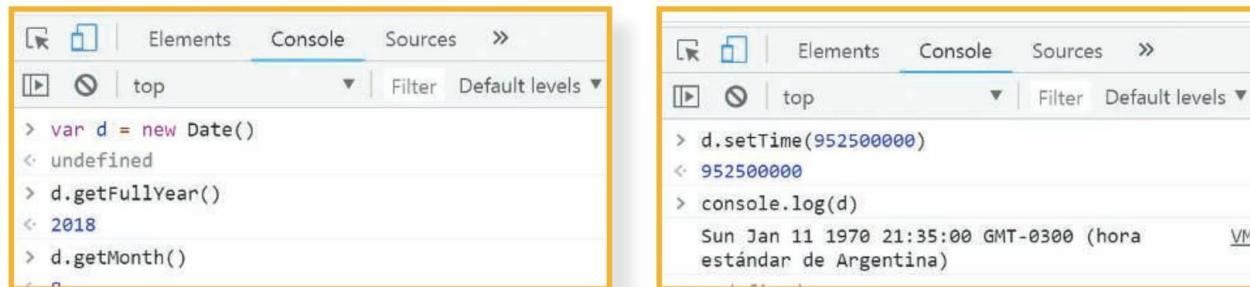
Métodos integrados en Date

En caso de que deseemos utilizar solo una parte de una fecha, podemos recurrir a los métodos integrados en el objeto **Date**. Estos nos facilitarán la tarea de extracción y nos ahorrarán mucho código. Veamos a continuación cuáles son:

MÉTODO	DESCRIPCIÓN	USO
getDay	Devuelve el número de día	d.getDay()
getMonth	Devuelve el número de mes, teniendo en cuenta que su conteo comienza en 0 (enero)	d.getMonth()
getYear	Devuelve dos dígitos del año	d.getYear()
getFullYear	Devuelve los cuatro dígitos del año	d.getFullYear()
getHours	Devuelve la hora	d.getHours()
getMinutes	Devuelve los minutos	d.getMinutes()
getSeconds	Devuelve los segundos	d.getSeconds()
getMilliseconds	Devuelve los milisegundos	d.getMilliseconds()
getTime	Devuelve la fecha en milisegundos, partiendo desde el 1 de enero de 1970	d.getTime()
getDay()	Devuelve el número de día de la semana, comenzando en 0 (domingo), hasta 6 (sábado)	d.getDay()

De la misma forma en que utilizamos métodos para obtener una determinada unidad de tiempo de **Date()**, también contamos con métodos que nos permiten establecer de manera fija una unidad de tiempo. Los vemos a continuación:

MÉTODO	DESCRIPCIÓN
setDate()	Establece el día como un número entre 1 y 31
setMonth()	Establece el mes como un número entre 0 y 11
setFullYear()	Establece el año como un número de cuatro dígitos
setMilliseconds()	Establece los milisegundos, entre 0 y 999
setHours()	Establece la hora entre 0 y 23
setMinutes()	Establece los minutos entre 0 y 59
setSeconds()	Establece los segundos entre 0 y 59
setTime()	Establece la fecha en milisegundos, partiendo desde el 1 de enero de 1970



The screenshot shows two instances of a browser's developer tools console. The left instance shows the following interactions:

```

> var d = new Date()
< undefined
> d.getFullYear()
< 2018
> d.getMonth()
< 0

```

The right instance shows the following interactions:

```

> d.setTime(952500000)
< 952500000
> console.log(d)
Sun Jan 11 1970 21:35:00 GMT-0300 (hora
estándar de Argentina)

```

/<Ejercicio práctico>

Calcular la edad de una persona

Veamos a continuación de qué forma podemos estimar la edad de una persona con JavaScript, teniendo el dato mínimo de su año de nacimiento. Abrimos la consola del navegador web para desarrollar el ejercicio allí. A continuación, creamos dos variables (**fi** y **fn**), donde **fi** almacena el año actual, y **fn** almacena el año de nacimiento de la persona:

```
var fi = new Date(2018)
var fn = new Date(1975)
```

Declaramos una variable denominada **edad**, en donde simplemente restamos al valor de **fi**, el año almacenado en **fn**. Luego, desplegamos mediante **console.log** el resultado almacenado en la variable **edad**:

```
var edad = (fi - fn);
console.log(edad);
```

/<Ejercicio práctico>

Visualizar la fecha actual en una web

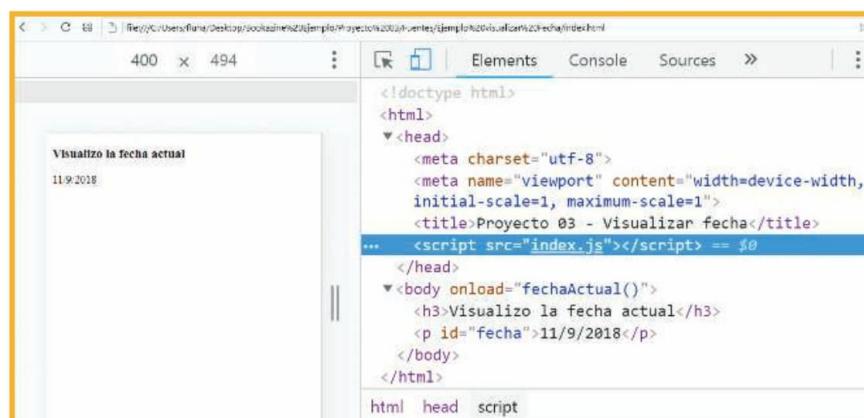
Construyamos a continuación el HTML base que usamos en cada proyecto, y también el archivo JS asociado. Dentro del archivo HTML, incluiremos un tag del tipo **<p>**, cuyo id sea “**fecha**”. Luego, en el archivo JS, creamos la función:

```
function fechaActual() {
    var d = new Date();
    document.getElementById("fecha").innerHTML =
    d.toLocaleDateString();
}
```

Solo nos queda adaptar el HTML, modificando la etiqueta **<body>** por:

```
<body onload="fechaActual()">
```

Veamos finalmente si el resultado de nuestro ejercicio es igual al de la imagen.



/<Ejercicio práctico>

Cuenta regresiva

A continuación realizaremos un ejercicio práctico que nos permitirá integrar en una web un sistema de cuenta regresiva, partiendo de la fecha actual, hasta una fecha futura determinada.



1 Como primera instancia, accedemos al portal de descargas del material que complementa esta obra. Ubicamos el archivo cuenta_regresiva.zip, y lo descargamos. En él encontraremos el HTML base ya escrito, que nos permitirá llevar a cabo la interfaz gráfica del proyecto.



2 Este proyecto utiliza Materialize CSS, un framework CSS y JS que complementa nuestro desarrollo con una interfaz gráfica moderna. Si editamos el archivo HTML, veremos que en el tag <HEAD> integra una hoja de estilos y un archivo de fuente tipográfica y, al final del tag <BODY>, un archivo JS; todos son archivos remotos.

 A screenshot of the VS Code interface showing the 'index.html' file. The code includes HTML structure, CSS imports for Materialize and icons, and a script tag pointing to 'index.js'. A specific line of CSS is highlighted: 'span#cuentaregresiva {color:white;text-align:center}<h3>111 d 50h 48 m 26s</h3>'.


```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1, maximum-scale=1">
    <title>Cuenta Regresiva</title>
    <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/materialize/1.0.0-beta.2/css/materialize.min.css" />
    <link href="https://fonts.googleapis.com/icon?family=Material+Icons" rel="stylesheet" />
    <script src="index.js"></script>
  </head>
  <body>
    <h3>Días restantes para el nuevo año ;)</h3>
    <div class="row"> <!-- item-width="100%" -->
      <div class="col s12 m10">
        <div class="card-panel red">
          <span id="cuentaregresiva" class="white-text"><h3>111 d 50h 48 m 26s</h3></span>
        </div>
      </div>
    </div>
    <script type="text/javascript" src="https://cdnjs.cloudflare.com/ajax/libs/materialize/1.0.0-beta.2/js/materialize.min.js"></script>
  </body>
```

3 En VS Code, creamos un nuevo proyecto denominado Cuenta regresiva y le agregamos el HTML incluido en el ZIP descargado. Lo editamos y antes del cierre de <HEAD> agregamos la referencia <SCRIPT> al archivo index.js. A continuación, dentro de este proyecto, creamos el archivo en cuestión.

 A screenshot of the VS Code interface showing the 'index.js' file. The code defines a 'countDownDate' variable as a Date object for January 1, 2019, at 00:00:00. It then uses a setInterval loop to calculate the difference between this date and the current date, and formats the result into days, hours, minutes, and seconds using Math.floor and modulus operations. A comment explains the use of Math.floor to convert milliseconds into readable time units.


```
var countDownDate = new Date("Jan 1, 2019 00:00:00").getTime();

// creamos una función integrada a la variable 'tiempo'
var tiempo = setInterval(function() {
  // Obtenemos el día actual
  var now = new Date().getTime();

  // Identificamos la diferencia de tiempo (en milisegundos) entre el día de hoy y nuestro
  var diferencia = countDownDate - now;

  /*
  Utilizamos Math.floor(), para convertir los milisegundos
  obtenidos en la variable 'diferencia'; en días - horas - minutos - segundos
  */
  var dias = Math.floor(diferencia / (1000 * 60 * 60 * 24));
  var horas = Math.floor((diferencia % (1000 * 60 * 60 * 24)) / (1000 * 60 * 60));
  var minutos = Math.floor((diferencia % (1000 * 60 * 60)) / (1000 * 60));
  var segundos = Math.floor((diferencia % (1000 * 60)) / 1000);

  // Mostramos el cálculo realizado en el id="cuentaregresiva"
  document.getElementById("cuentaregresiva").innerHTML = " <h3>" + dias + " d " + horas + " h " + minutos + " m " + segundos + " s</h3>";
}, 1000);
```

4 Comenzamos entonces a escribir el código en el archivo index.js. Lo que básicamente hará este script es definir las variables a utilizar, definir la fecha futura para la cual queremos calcular el tiempo faltante, obtener el tiempo restante entre hoy y la fecha futura, y convertir los milisegundos del tiempo restante en una fecha legible por nosotros.

Código JavaScript

Comenzamos por declarar el objeto **Date()** que almacenará la fecha futura:

```
var countDownDate = new Date("Jan 1, 2019 00:00:00").getTime();
```

A continuación, creamos una variable denominada **tiempo**, la cual contendrá la función JS **setInterval()**.

```
// Creamos una función integrada en la variable 'tiempo'  
var tiempo = setInterval(function() {...  
}, 1000);
```

setInterval()
se ejecutará
cada 1 segundo.

Ahora reemplazemos los tres puntos dentro del código anterior, por el siguiente código:

```
var now = new Date().getTime();  
var diferencia = countDownDate - now;
```

En la primera línea, obtenemos el día actual.

En la siguiente, identificamos la diferencia de tiempo (en milisegundos) entre el día de hoy y nuestro día futuro.

A continuación, pasamos a los cálculos matemáticos:

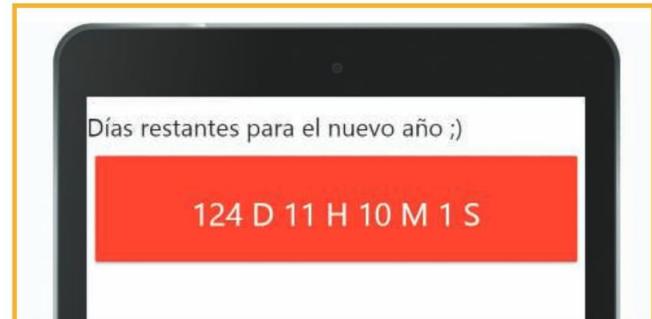
```
var dias = Math.floor(diferencia / (1000 * 60 * 60 * 24));  
var horas = Math.floor((diferencia % (1000 * 60 * 60 * 24)) / (1000 * 60 * 60));  
var minutos = Math.floor((diferencia % (1000 * 60 * 60)) / (1000 * 60));  
var segundos = Math.floor((diferencia % (1000 * 60)) / 1000);
```

Creamos las variables **días**, **horas**, **minutos** y **segundos**, y utilizamos la función **Math.floor()**, que nos ayuda a calcular cada uno de los valores para ellas, pasándole a cada operación la variable **diferencia**, la cual almacena el resultado entre la

fecha de hoy y la fecha futura (en milisegundos). El método **floor()** del objeto **Math** se ocupa de redondear el número que da como resultado cada una de estas operaciones. El código final es el siguiente:

```
document.getElementById("cuentaregresiva").innerHTML = "<h3>" + dias + " D " +  
horas + "H " + minutos + " M " + segundos + " S " + "</h3>";  
if (distancia < 0) {  
    clearInterval(tiempo);  
    document.getElementById("cuentaregresiva").innerHTML = "<h3>¡Feliz 2019  
para todos!</h3>";
```

Finalmente, mostramos el resultado obtenido en el elemento "**cuentaregresiva**", y si la fecha futura es menor que la actual, mostramos al usuario un saludo u otro tipo de mensaje.



Arreglos



Conozcamos qué son los arreglos y cuáles son los principales métodos y funciones que los complementan para que, al momento de programar, se simplifique el acceso a la información que almacenan.

Utilizar arreglos en JS es un procedimiento eficaz para mantener ordenados y a mano aquellos pequeños bloques de información que necesitamos manipular o visualizar, y así

no tener que acceder a una base de datos o archivo JSON cuando precisamos manipularlos. Veamos a continuación una descripción rápida de los arreglos:

```
var lenguajes = ['Visual C#', 'Objective C',
    'C++', 'Python', 'R', 'Javascript', 'Kotlin'];
```

Declaramos los arreglos en JS de la misma forma que una variable.

Para contener los elementos que conformarán el arreglo, abrimos y cerramos corchetes, y agrupamos el contenido dentro de ellos.

Si el contenido del arreglo está compuesto por cadenas de texto, por cada elemento debemos abrir comillas (simples o dobles) y cerrarlas al finalizar cada ítem.

Para almacenar textos, las comillas (simples o dobles) deben ser consistentes. Por una cuestión de prolifidad, hay que usar el mismo tipo de comilla por cada nuevo elemento.

Si tenemos que almacenar números dentro del arreglo, en vez de cadenas, podemos obviar la apertura y el cierre de comillas.

Todos los elementos que conforman el arreglo deben estar separados por coma.

Crear arreglos

Realicemos el primer ejercicio manipulando arreglos creados por nosotros mismos. Abrimos el navegador web y accedemos a las herramientas para el desarrollador. Nos posicionamos en la consola, donde escribimos las siguientes líneas de código:

```
var fibonacci = [8, 13, 2, 34, 5, 0, 1, 21, 3];
var lenguajes = ['Visual C#', 'Javascript', 'Python', 'C++', 'PHP', 'Kotlin'];
```

En estas dos líneas de código hemos creado dos arreglos en JavaScript: uno de cadena numérica y otro de texto. Veamos a continuación las propiedades y funciones más comunes que nos permiten sacar partido de estos arreglos rápidamente.

```
> var fibonacci = [8, 13, 2, 34, 5, 0, 1, 21, 3];
< undefined
> var lenguajes = ['Visual C#', 'Javascript', 'Python', 'C++', 'PHP', 'Kotlin'];
< undefined
>
```

Métodos más comunes para el manejo de arreglos

FUNCIÓN O MÉTODO	DESCRIPCIÓN
arreglo.length()	Obtiene el total de elementos que componen el array.
arreglo[i]	[i] corresponde al número de índice. Devuelve el ítem correspondiente almacenado en ese índice.
arreglo.indexOf(item)	ítem corresponde a la cadena del array. Devuelve el número de índice donde está alojado ese elemento.
arreglo.sort()	Ordena el total de elementos almacenados en el array.
arreglo.push(nuevoelemento)	Agrega un elemento en el final del arreglo.
arreglo.pop()	Elimina el último elemento del final del array.
arreglo.shift()	Elimina el primer elemento contenido en el array.
arreglo.unshift(nuevoelemento)	Agrega un elemento al principio del array.

Probemos algunos de estos métodos sobre los arreglos que hemos creado.

Volvamos a la consola del navegador web y escribamos la siguiente línea:

```
fibonacci.length
```

Veremos en la consola un valor numérico, que para el arreglo denominado **fibonacci** será el número **9**. Notemos que los arreglos inician la indexación de su contenido a partir del **0** (cero). Este último dato es el que debemos

```
fibonacci[7]
```

tener en cuenta cuando deseemos acceder a un ítem del arreglo en particular. Por ejemplo, si queremos acceder al elemento 5 del arreglo, entonces tendremos que escribir:

Según cómo posicionamos los números al momento de crear el arreglo, el 5 fue ubicado en la posición 7 (comenzando el conteo desde 0).

Ordenemos a continuación este arreglo, utilizando el método correcto, según lo que especificamos en la tabla anterior:

```
fibonacci.sort()
```

Obtendremos así en pantalla el orden correspondiente de los números que componen este array.

```
fibonacci = [8, 13, 2, 34, 5, 0, 1, 21, 3];
fibonacci.length
9
fibonacci.sort()
(9) [0, 1, 13, 2, 21, 3, 34, 5, 8]
|
```

Ubicar y agregar elementos

Trabajemos ahora con el otro arreglo creado. En él incluimos diversos lenguajes de programación de forma desordenada. Busquemos entonces el índice correspondiente al lenguaje JavaScript, como se muestra a continuación:

```
lenguajes.indexOf('Javascript');
```

Ahora agreguemos un nuevo lenguaje a la lista:

```
lenguajes.push('GoLang');
```

Como podemos ver, al agregar un nuevo ítem al arreglo, este se redimensiona.

Comprobemos esto validando todos los índices del arreglo y listando el último de ellos.



The screenshot shows the browser's developer tools with the 'Console' tab selected. The console output is as follows:

```
var lenguajes = ['Visual C#', 'Javascript', 'Python', 'C++', 'PHP', 'Kotlin'];
< undefined
> lenguajes.indexOf('Javascript');
< 1
> lenguajes.push('GoLang');
< 7
> lenguajes.length
< 7
> lenguajes[6]
< "GoLang"
>
```

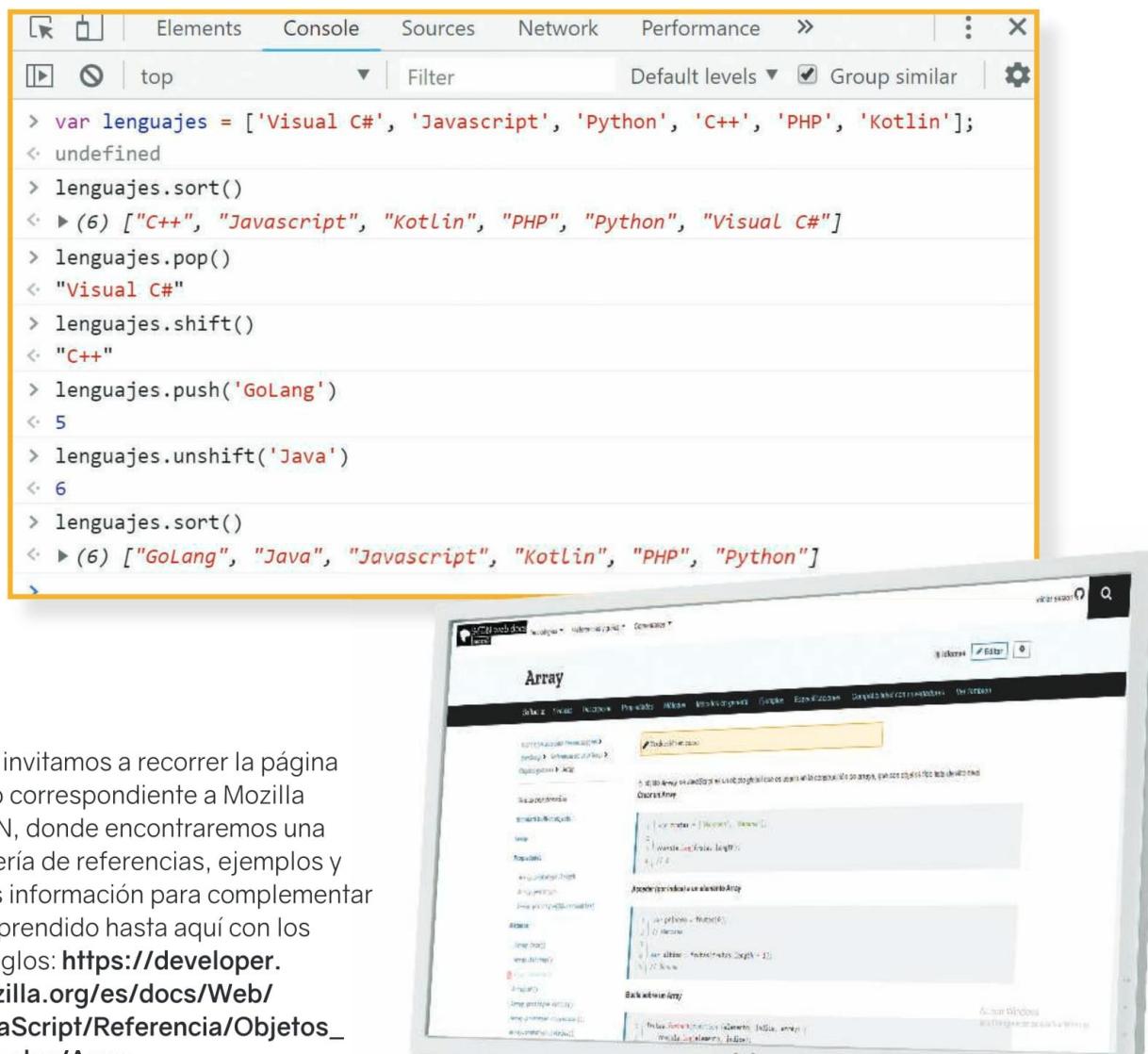


Práctica con arreglos

A continuación, realizaremos una práctica sobre la consola del navegador web, con los arreglos que hemos creado. En principio, trabajaremos con el arreglo **lenguajes**, en el que debemos realizar lo siguiente:

- Regenerar el arreglo
- Ordenarlo
- Eliminar el primer elemento del arreglo
- Eliminar el último elemento del arreglo
- Agregar al principio del arreglo un nuevo lenguaje de programación
- Agregar otro lenguaje de programación al final del arreglo
- Por último, volver a ordenar el arreglo

En la siguiente imagen, podemos apreciar el resultado de este ejercicio, habiendo agregado los lenguajes de programación **GoLang** y **Java**.



The screenshot shows a browser's developer tools console tab. The code entered and its output are as follows:

```
var lenguajes = ['Visual C#', 'Javascript', 'Python', 'C++', 'PHP', 'Kotlin'];
< undefined
> lenguajes.sort()
< ▶ (6) [ "C++", "Javascript", "Kotlin", "PHP", "Python", "Visual C#" ]
> lenguajes.pop()
< "Visual C#"
> lenguajes.shift()
< "C++"
> lenguajes.push('GoLang')
< 5
> lenguajes.unshift('Java')
< 6
> lenguajes.sort()
< ▶ (6) [ "GoLang", "Java", "Javascript", "Kotlin", "PHP", "Python" ]
>
```

Below the console, a screenshot of the Mozilla Developer Network (MDN) documentation page for the **Array** object is visible. The page includes the MDN logo, a search bar, and several sections of text and code examples related to arrays.

Los invitamos a recorrer la página web correspondiente a Mozilla MDN, donde encontraremos una batería de referencias, ejemplos y más información para complementar lo aprendido hasta aquí con los arreglos: https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array

Cadenas de texto



Repasemos de qué forma podemos mejorar el manejo de cadenas de texto almacenadas en variables, accediendo a las propiedades y métodos más útiles que contiene cada variable JS.

En el inicio de esta obra, dimos un panorama general sobre el uso de variables. Ahora, profundizaremos en el tema, explorando algunos métodos de las variables que nos permiten explotar un poco más su funcionalidad desde nuestra perspectiva de programadores. Conozcamos los principales métodos a través de la siguiente tabla:



MÉTODOS	DESCRIPCIÓN
<code>variable.length()</code>	Obtiene el total de elementos que componen el array.
<code>indexOf(cadena)</code>	(cadena) corresponde al texto que se quiere ubicar. El método <code>indexOf</code> devuelve el índice correspondiente donde inicia dicha cadena de texto.
<code>search(cadena)</code>	El método <code>search()</code> permite buscar una cadena de texto dentro del texto contenido en esta variable, retornando la posición inicial del resultado.
<code>substr(i, j)</code>	Estos parámetros funcionan casi de igual manera, cortando una porción de la cadena de texto, de acuerdo con los parámetros establecidos mediante los índices [i] y [j].
<code>slice(i, j)</code>	Existen diferencias entre sí, que veremos en la siguiente infografía.
<code>replace(a, b)</code>	Este método permite reemplazar un bloque de texto dentro de una cadena, por otro bloque de texto. El parámetro [a] es el bloque que queremos buscar, mientras que el parámetro [b] contiene la cadena de texto a reemplazar.
<code>concat()</code>	Permite concatenar o unir dos fragmentos de texto.
<code>toUpperCase()</code>	Convierte en mayúsculas la cadena de texto contenida en la variable.
<code>toLowerCase()</code>	Convierte en minúsculas la cadena de texto contenida en la variable.
<code>trim()</code>	Elimina los espacios en blanco al inicio o fin de una cadena de texto.
<code>charAt(i)</code>	Extrae el carácter correspondiente a la posición indicada a través del parámetro [i].
<code>split(j)</code>	Convierte una cadena de texto en una variable indexada, separando su contenido a través de un carácter comodín, indicado a través del parámetro [j].

Veamos en la siguiente imagen un ejemplo de cómo transformar en mayúsculas y minúsculas la cadena de texto almacenada en una variable:

```
Elements Console Sources Network Performance > ... X
top Filter Default levels Group similar 1 hidden | 1 message
No user mess...
No errors
No warnings
No info
1 verbose
> var texto = "Esto es Javascript";
< undefined
> texto.toUpperCase();
< "ESTO ES JAVASCRIPT"
> texto.toLowerCase();
< "esto es javascript"
>
```

Marcar diferencias entre métodos

Como pudimos apreciar en la tabla anterior, algunos métodos incluidos en las variables poseen distinto nombre, pero cumplen casi la misma función. ¿Hay una diferencia sustancial entre ellos? Sí, la hay, y la mostramos en la siguiente infografía.

```
var texto = 'Esto es un fragmento de texto.';
```

La variable texto almacena una cadena de caracteres.

Esta cadena de caracteres posee un total de 30 posiciones (0 al 29).

```
var sst = texto.substr(0, 20);
```

El método **substr()** (sustraer) extrae de la variable texto, un fragmento de la cadena, comenzando por la posición 0 (cero); un total de 20 (veinte) caracteres. Y lo almacena en la variable **sst**.

```
var str = texto.substring(11, 20);
```

El método **substring()** (sustraer una cadena) extrae de la variable texto un fragmento de la cadena de caracteres, desde la posición 11 (once) hasta la posición 20 (veinte).

Podemos obviar el último parámetro de este método, y así obtendremos la cadena de caracteres, hasta el final.

```
var sst = texto.slice(11, 29);
```

El método **slice()** corta el contenido de la variable texto, entre la posición inicial y final indicada. A diferencia de **substring()**, hay que especificar ambos parámetros de forma obligatoria.

Práctica rápida

Ejercitemos en la consola del navegador web lo visto en la infografía anterior, para tener una mejor idea de cómo se comporta cada uno de los métodos al momento de utilizarlos.

Verifiquemos qué resultado obtendremos si implementamos una variable negativa como parámetro en el método `slice()`, por ejemplo:

```
texto.slice(-19);
```

```

▼ | Filter Default levels ▾ Group similar 2 hidden
> var texto = 'Esto es un fragmento de texto.';
< undefined
> texto.slice(11,29)
< "fragmento de texto"
> texto.substr(8)
< "un fragmento de texto."
> texto.substring(11, 20)
< "fragmento"
> texto.slice(-19)
< "fragmento de texto."
> |

```

Concatenar textos

Veamos a continuación cómo concatenar o unir dos textos diferentes, utilizando la función `concat()`. Para hacerlo, volvemos a la consola del navegador web y creamos las siguientes variables con sus respectivos valores:

```
var nombre = "Fernando";
var apellido = "Luna";
```

Ya tenemos dos variables, una con el nombre y otra con el apellido.

Ahora crearemos una tercera variable donde concatenaremos ambos datos:

```
var nombrecompleto = nombre.concat(" " + apellido);
```

Como podemos ver, debemos utilizar una de las dos variables (`nombre` es el caso ideal) para concatenar la segunda variable (`apellido`). En este caso, debemos destacar que el método `concat()` puede recibir múltiples parámetros, no se limita solamente a dos:

```
nombres.concat("Fernando", " Nicolás ", " Julián ", " Omar ")
```

```

Elements Console Sources Network Performance »
top Filter Default levels ▾ Group similar G
> var nombre = "Fernando";
< undefined
> var apellido = "Luna";
< undefined
> var nombrecompleto = nombre.concat(" " + apellido);
< undefined
> nombrecompleto
< "Fernando Luna"

```

Separar cadenas de texto

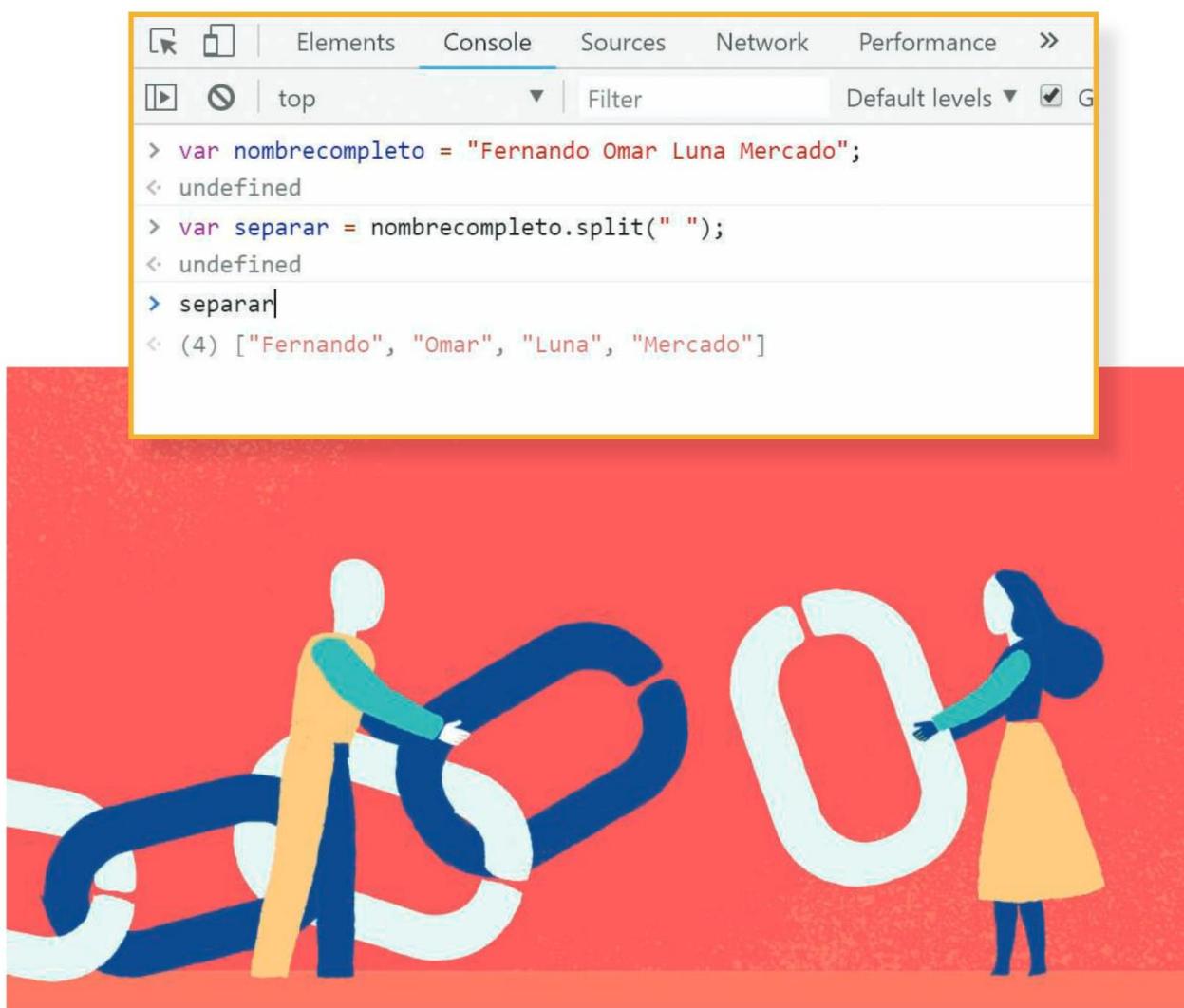
De la misma manera en que podemos concatenar, también podemos separar. Para hacerlo, utilizamos el método **split()**, que se ocupa de separar el contenido de una cadena de texto, o números, tomando como parámetro un carácter especial o comodín. Veamos a continuación un ejemplo:

```
var nombrecompleto = "Fernando Omar Luna Mercado";
```

Tenemos una variable conformada por un nombre completo, donde el único carácter que podemos utilizar como comodín es el espacio entre cada nombre. Entonces, creamos una nueva variable, que obtendrá el resultado de la separación de palabras:

```
var separar = nombrecompleto.split(" ");
```

Ahora, ¿cuál es el resultado para acceder a cada uno de los nombres? La variable que recibe la cadena separada crea un índice por cada una de las palabras. Así, podremos acceder a cada palabra simplemente utilizando el nombre de la cadena, más el índice.



/<Ejercicio práctico>

Reemplazo de marcadores

Veamos a continuación cómo aprovechar las capacidades de búsqueda y reemplazo de cadenas de caracteres, creando un sistema que nos permita ingresar un texto que queremos buscar, y otro que oficiará como reemplazante del primero.



- 1 Descargamos de la web correspondiente a esta obra el archivo .ZIP denominado **Proyecto04_ReemplazarMarcadores**. En él encontraremos un archivo HTML y otro TXT. Creamos un nuevo proyecto llamado Reemplazo de marcadores, y agregamos ambos archivos a la carpeta.

```

<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no" />
    <title>Reemplazar Marcadores</title>
    <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/@materialize-css/materialize@1.0.0/dist/css/materialize.min.css" />
    <script src="index.js"></script>
  </head>
  <body>
    <h3>Reemplazar Marcadores</h3>
    <div class="row">
      <div class="col s12 m10">
        <div class="card-panel light-blue">
          <textarea id="texto" class="materialize-textarea"></textarea>
        </div>
      </div>
      <label for="txtbuscar">Palabra a buscar:</label>
      <input type="text" id="txtbuscar"/>
      <label for="txtreemplazar">Reemplazar por:</label>
      <input type="text" id="txtreemplazar"/>
    </div>
  </body>
</html>

```

- 2 Ahora creamos dentro del proyecto un archivo .JS. Luego abrimos el archivo HTML, y dentro del apartado <head>, ubicamos el espacio correspondiente antes del cierre de este tag, donde incluimos la referencia al JS creado.

```

var b;
var r;
var t;

function limpiarCampos() {
  document.getElementById("txtbuscar").value = "";
  document.getElementById("txtreemplazar").value = "";
}

function reemplazar() {
  b = document.getElementById("txtbuscar").value;
  r = document.getElementById("txtreemplazar").value;
  t = document.getElementById("texto").value;
  if (b.trim().length > 0 & r.trim().length > 0) {
    if (t.length > 0) {
      var cambio = t.replace(b, r);
      document.getElementById("texto").value = cambio;
    }
  }
}

```

- 3 El siguiente paso es agregar el código funcional dentro del archivo index.js. En él declaramos tres variables, con las cuales manejaremos las cadenas de caracteres; y dos funciones: **limpiarCampos()** y **reemplazar()**. Veamos el código:

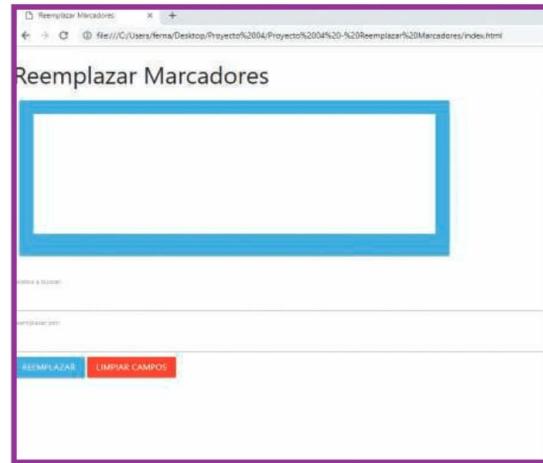
```

var b;
var r;
var t;

```

Simplificamos en tres variables (**b**, **r**, **t**) la manipulación de cadenas de texto que haremos en este ejercicio. La primera almacenará el marcador que debemos ubicar en el texto, la segunda almacenará la palabra que reemplazará el marcador buscar y, finalmente, la tercera contendrá el bloque de texto en cuestión.

- 4** Como podemos apreciar, en la interfaz de nuestra aplicación web contamos con un campo que tiene múltiples líneas de texto, dos cajas de texto adicionales (Buscar y Reemplazar) y dos botones: Reemplazar y Limpiar campos. Agregamos el código para limpiar los campos:

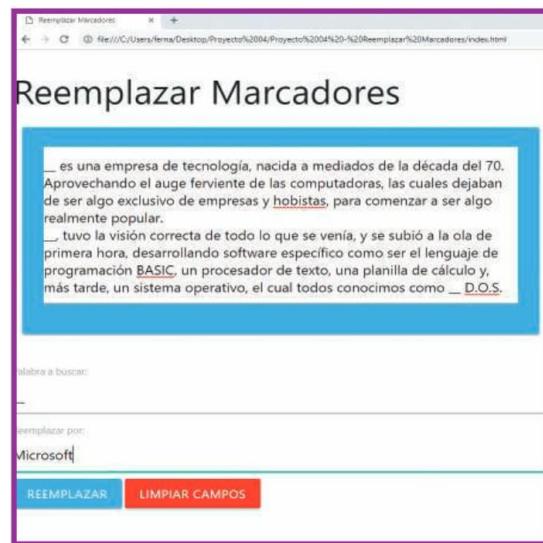


```
function limpiarCampos() {  
    document.getElementById("txtbuscar").value = "";  
    document.getElementById("txtreemplazar").value = "";  
}
```

Vemos que la función **limpiarCampos()** simplemente se ocupa de vaciar el contenido agregado en las cajas de texto identificadas por los nombres **txtbuscar** y **txtreemplazar**.

```
function reemplazar() {  
    b = document.getElementById("txtbuscar").value;  
    r = document.getElementById("txtreemplazar").value;  
    t = document.getElementById("texto").value;  
    if (b.trim().length > 0 && r.trim().length > 0) {  
        if (t.length > 0) {  
            var cambio = t.replace(b, r);  
            document.getElementById("texto").value = cambio;  
        }  
    }  
}
```

- 5** La función **reemplazar()** captura en las variables **b**, **r** y **t** los valores de los campos correspondientes. Mediante **if()** analizamos si las variables **b** y **r** contienen datos cargados. Finalmente, analizamos si la variable **t** contiene una cadena de datos y, por último, ejecutamos el método **replace()**, pasándole las funciones **b** (buscar) y **r** (reemplazar). Para terminar, ubicamos la función **onclick=""** de los botones, y les asignamos las funciones JS correspondientes. Probamos el resultado del proyecto pegando, en el campo **textarea**, el bloque de texto almacenado en el archivo TXT.



Formularios y datos



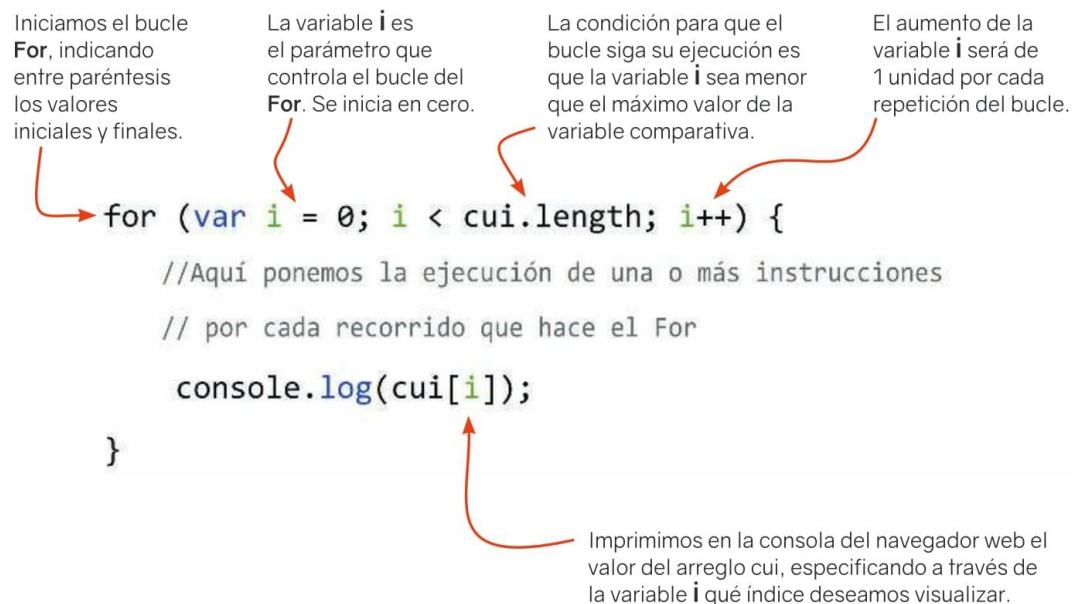
Repasemos los diferentes tipos de datos que se pueden manejar en formularios HTML, conozcamos cómo la sentencia **For** nos ayuda a cargar información en campos de selección, y aprendamos a validar datos en formularios.

La sentencia de bucle **For** nos ayuda a repetir un bloque de código específico **N** cantidad de veces, tomando como premisa un valor finito para **N**. Uno de los usos efectivos de esta sentencia es recorrer arreglos, listando los elementos en

pantalla, o dentro de algún componente web, como un combo de selección. Para comprender bien su funcionamiento, vamos a abrir la consola del navegador web y a escribir en ella el código del siguiente arreglo:

```
var cui = ["China", "Indu", "Italiana", "Centroamericana", "Mexicana",  
"Mediterranea"];
```

Antes de poner en práctica este bucle, tengamos presente el nombre de este arreglo, y repasemos la siguiente infografía:



Escribamos ahora en la consola del navegador el siguiente bucle **For**, para poder evaluar su resultado:

```
for (var i = 0; i < cui.length; i++) { console.log(cui[i]); }
```

```

> var cui = ["China", "Indu", "Italiana", "Centroamericana",
"Mexicana", "Mediterranea"];
< undefined
> for (var i = 0; i < cui.length; i++) { console.log(cui[i]); }
China
Indu
Italiana
Centroamericana
Mexicana
Mediterranea
< undefined

```

The screenshot shows the browser's developer tools with the 'Console' tab selected. It displays a JavaScript for loop that iterates through an array named 'cui'. The array contains six strings: 'China', 'Indu', 'Italiana', 'Centroamericana', 'Mexicana', and 'Mediterranea'. The loop logs each element to the console, and the output is shown on the right with file references like 'VM6964:1'.

Como podemos notar, este recorre el arreglo cui, e imprime en la consola el valor de cada uno de sus índices.

Ejercitación del bucle For

Existen diversos tipos de campos que permiten el ingreso de datos a través de formularios web. En esta oportunidad repasaremos cuáles son y qué podemos hacer desde JavaScript para controlar y validar el ingreso de datos.

Pero, antes de empezar, veamos con un ejemplo real cómo aprovechar el bucle **For** para cargar un campo de formulario del tipo **Select**.

Llevaremos a la práctica el ejemplo mencionado anteriormente, para lo cual necesitamos descargar de la web oficial de esta obra el archivo **Proyecto05_sentenciaFor.zip**. Lo descomprimimos y creamos un nuevo proyecto en VS Code agregando el documento HTML incluido en el .ZIP.



The screenshot shows the VS Code interface with two tabs: 'index.html' and 'JS Index.js'. The 'index.html' tab contains the following code:

```

1  <!DOCTYPE html>
2  <html>
3      <head>
4          <meta charset="utf-8" />
5          <title>Javascript - Sentencia For</title>
6          <meta name="viewport" content="width=device-width, initial-scale=1.0" />
7      </head>
8      <body>
9          <h2>Sentencia For</h2>
10         <div id="etiqueta" item-width="80px">
11             <label for="tipo-de-cocina">Tipo de cocina preferida:</label>
12             <select id="tipo-de-cocina">
13                 <option id="Todas">Todas</option>
14             </select>
15         </div>
16         <script src="index.js"></script>
17     </body>
18 </html>

```

The 'preview' tab shows a user interface with a dropdown menu titled 'Sentencia For' containing the option 'Todas'.

Si editamos el archivo HTML, veremos que contiene un elemento del tipo **Select**. Este despliega en el documento web un componente similar a un Combo Box, que puede incluir

múltiples opciones para que el usuario seleccione una. El componente cuenta con un **<tag>** que permite agregar más de una opción para que esta sea listada por el componente.

```

<select id="tipo-de-cocina">
    <option id="Todas">Todas</option>
    <option id="...">otra opción</option>
    <option id="...">otra opción más</option>
</select>

```

Como podemos ver en el código correspondiente al componente **Select**, este cuenta con un ID (**tipo-de-cocina**) y con una serie de tags **<option>** que incluyen un ID y una descripción. La descripción hace referencia al dato "amigable" que visualiza el usuario en pantalla cuando despliega el componente. El ID del tag puede referirse a un ID numérico o a un texto similar al dato visualizado en pantalla. En nuestro ejemplo apuntaremos a esto último.

Agregar opciones dinámicamente

Ya con el proyecto creado, agregamos el archivo **index.js** y lo declaramos dentro del archivo HTML. En el archivo JS declaramos la variable **cui**, tal como lo hicimos al inicio de esta sección.

```
var cui = ["China", "Indu", "Italiana", "Centroamericana", "Mexicana",  
"Mediterranea"];
```

A continuación de este código, declaramos la siguiente sentencia:

```
var seleccion = document.getElementById('tipo-de-cocina');
```

La variable **seleccion** no es una variable en sí, sino que en este caso está declarada como un objeto. ¿Qué objeto? El objeto **Select** propio del componente HTML, que nos será útil para agregar de forma dinámica cada uno de los elementos contenidos en el arreglo **cui**, utilizando, por supuesto, el bucle **For**. Agreguemos entonces el código correspondiente al bucle:

```
for(var i = 0; i < cui.length; i++) {  
    var opt = document.createElement('option');  
    opt.innerHTML = cui[i];  
    opt.value = cui[i];  
    seleccion.appendChild(opt);  
}
```

Como podemos ver en el código correspondiente al bucle **For**, recorremos cada uno de los elementos que componen el arreglo **cui**. Cuando entramos en el bucle por cada elemento, creamos un objeto del tipo **option** directamente en la variable **opt**.

Mediante **innerHTML**, escribimos el texto del elemento seleccionado, y en la propiedad **value**, repetimos dicho texto, que oficiará como índice.



A través del método **appendChild()** del objeto **seleccion**, agregamos el elemento a mostrar en pantalla y el elemento que oficia de índice al tag **option**. Este bloque de código se ejecutará por cada índice que contiene el arreglo **cui**.

Modifiquemos a continuación el código de nuestro proyecto, incluyendo en él dos sentencias del tipo `console.log()`, donde una de ellas liste el valor de la variable `i` por cada repetición del bucle **For**, y la otra liste en la consola el valor del texto a visualizar y el valor correspondiente al valor impuesto al tag **option**.

El resultado obtenido debe ser similar al que vemos en la siguiente imagen:

```

504 x 490
Elements Console Sources > □ top Filter Default levels □
Valor del indice: 0 index.js:10
China China index.js:13
Valor del indice: 1 index.js:10
Indu Indu index.js:13
Valor del indice: 2 index.js:10
Italiana Italiana index.js:13
Valor del indice: 3 index.js:10
Centroamericana Centroamericana index.js:13
Valor del indice: 4 index.js:10
Mexicana Mexicana index.js:13
Valor del indice: 5 index.js:10
Mediterranea Mediterranea index.js:13
> |
```

Tipos de datos en formularios

En HTML existen diferentes tipos de datos que pueden ser ingresados a través de los elementos `<input>`. Los más comunes son los del tipo texto y numérico, pero con la llegada de HTML5, esta capacidad se amplió notablemente, gracias al soporte que nació para facilitar el ingreso de datos en los formularios complejos y, a su vez, para aprovechar las capacidades diversas que nos proponen los teclados de los dispositivos móviles.

Veamos a continuación una tabla con cada tipo de dato soportado por los `<input>`:

Envíenos sus comentarios

Nombre _____ Apellido _____

Correo electrónico _____

Indique su sitio web _____

Indique su número de teléfono _____

Año de nacimiento
1975 _____

ACEPATAR ➤ **CANCELAR**

VALOR	DESCRIPCIÓN
Color	Inicia una paleta de colores.
Date	Define el campo tipo fecha.
datetime-local	Muestra el campo date + time (sin especificar la zona horaria).
Email	Acepta contenido del tipo dirección de correo electrónico.
Month	Permite ingresar mes y año.
Number	Acepta solo datos numéricos.
Range	Visualiza un control Slider, limitado por un valor inicial y uno final.
Search	Muestra un campo alfanumérico, con un botón de borrar contenido.
Tel	Permite ingresar solo números telefónicos.
Text	El campo alfanumérico por defecto, o convencional.
Time	Acepta el ingreso de horas, minutos y segundos.
Week	Permite ingresar el número de semana de un año.

Pero ¿por qué tenemos que conocer algo que corresponde a HTML? Es importante porque, en el mundo profesional, todo este tipo de desarrollos se hace directamente desde JavaScript. Tal como vimos en el ejercicio donde explicamos el bucle **For**, creando desde código un input type, en el mundo del desarrollo de software con JS puro o a través de algún framework, las interfaces gráficas de aplicaciones web o móviles se crean íntegramente con JS.

Envíenos sus comentarios	
Nombre	HTML5: <input type="text"> JS: document.createElement('text');
Apellido	
Correo electrónico	HTML5: <input type="email"> JS: document.createElement('email');
Indique su sitio web	
Indique su número de teléfono	HTML5: <input type="tel"> JS: document.createElement('tel');
Año de nacimiento	
1975	HTML5: <input type="date"> JS: document.createElement('date');
<input style="background-color: green; color: white; border: none; padding: 5px 10px; margin-right: 10px;" type="button" value="ACEPTAR"/> <input style="background-color: red; color: white; border: none; padding: 5px 10px;" type="button" value="CANCELAR"/>	

En esta infografía podemos ver cómo se crea cada tipo de campo en HTML5, así como también su equivalente en código JavaScript.

Beneficios para las plataformas móviles

Principalmente, el cambio que hubo en los input types a partir de HTML5 produjo un beneficio notorio para quienes desarrollan aplicaciones web, más que nada en el terreno mobile. Para entender mejor este tema, accedamos al material complementario que acompaña a esta obra y descarguemos el archivo **Ejemplo Formulario de Datos.zip**. Lo descomprimimos y copiamos el HTML a un servidor web en Internet o dentro de la red local de nuestro hogar.

La idea es visualizar esta página web desde un dispositivo móvil, ya sea teléfono inteligente o tablet. Una vez cargada la página en el dispositivo, hagamos tap sobre el **campo web** o el **campo email**. Veremos que el tipo de teclado virtual del dispositivo se adapta para permitir el ingreso específico del tipo de dato indicado por el campo.

Lo mismo ocurre si hacemos clic en el campo **Teléfono**. En este caso, se debe activar, principalmente en los teléfonos móviles, el teclado apto para ingresar solo números o simbología acorde (# * -).



Validar por código el contenido del formulario

Realicemos un ejercicio que nos permita aprovechar el contenido del formulario de ejemplo descargado, para validar en él cada uno de los campos que incluye.

El primer paso es crear el archivo index.js que no está incluido en el proyecto. Luego, verificamos que el archivo HTML tenga la referencia correspondiente al archivo JavaScript.

Finalmente, analizaremos el código necesario para que este formulario valide el contenido de sus campos, antes de guardarlo.



Funciones de retorno

Creamos una función JS por cada uno de los campos a validar. ¿Qué es una función de retorno? Se trata de una función que tiene una expresión comparativa en su interior y que, sobre la base del resultado de la comparación, retorna TRUE o FALSE a través de la misma función. Veamos un ejemplo rápido:

```
function EsUnaPrueba() {  
    if (1 == 1) {  
        return TRUE;  
    } else {  
        Return FALSE;  
    }  
}
```

¿Qué ocurre en esta función? Comparamos mediante **If** si el valor **1** es igual a **1**.

Como esta igualdad es correcta o verdadera, entonces se ejecuta **return True**.

En caso contrario, el valor de la función es **falso**. Ahora, para validar qué resultado devuelve esta función, escribimos el siguiente código en otro bloque JS:

```
If (EsUnaPrueba) {  
    //La comparación devuelve VERDADERO  
    //Ejecutar acá el código correspondiente  
    ...  
}
```

Expresiones regulares en JS

Una expresión regular es, básicamente, un patrón determinado de textos o caracteres que nos permiten encontrar (o no) coincidencia de uno o más de ellos dentro de una variable o bloque de texto. Por lo general, estos patrones se utilizan dentro de JS a través de los métodos **Exec**, **Test**, **Regexp**, **Match**, **Replace** y **Search**.

En nuestra validación de campos utilizaremos

principalmente el método **Test**, en contraste con el contenido cargado en el campo con un bloque de expresión regular que nos indicará los valores que debemos analizar.

Los campos que analizaremos de este formulario son: Nombre, Apellido, Correo electrónico, Sitio web y número de teléfono. Veamos entonces la primera sentencia dentro del archivo JS.

Comparar el contenido de los campos

```
JS index.js
1  function nombre_es_string() {
2      var nombrestring = document.getElementById('txtnombre').value;
3      if (isNaN(nombrestring)) {
4          return true;
5      } else {
6          return false;
7      }
8
9
10 function apellido_es_string() {
11     var apellidostring = document.getElementById('txtapellido').value;
12     if (isNaN(apellidostring)) {
13         return true;
14     } else {
15         return false;
16     }
17 }
```

1 Aquí podemos apreciar el código de dos funciones, para validar nombre y apellido. Capturamos en una variable el valor de cada campo, y mediante la función `isNaN()`, verificamos que dicho campo no tenga ingresado un valor numérico. Si da como resultado **TRUE**, entonces damos por válido que el contenido del campo es texto.

```
19 function es_url() {
20     var expresion = /(http|https):\/\/((\w+:{0,1}\w*)?(\$+)(:[0-9]+)?(\/|\|( [\\w#!:.?+=&%!~-\\/]))?/;
21     var urlstring = document.getElementById('txturl').value;
22     if (expresion.test(urlstring)) {
23         return true;
24     } else {
25         return false;
26     }
27
28
29 function es_email() {
30     var expresion = /^[A-Za-z0-9._%-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4}$/;
31     var emailstring = document.getElementById('txtemail').value;
32     if (expresion.test(emailstring)) {
33         return true;
34     } else {
35         return false;
36     }
37 }
```

2 Para comparar una URL y una dirección de correo electrónico, usamos dos bloques de expresiones regulares, que permiten validar los caracteres que acepta cada uno de estos tipos de campos. La URL deberá contener HTTP o HTTPS además de caracteres válidos, y la dirección de e-mail deberá incluir @, entre el resto de caracteres soportados. Luego utilizamos el método `test()` para comparar y aceptar o rechazar su contenido válido.

```
39 function es_telefono() {
40     var expresion = /^[+]*[(]{0,1}[0-9]{1,3}[)]{0,1}[-\s./0-9]*$/g;
41     var nrotelefono = document.getElementById('txttelefono').value;
42     if (expresion.test(nrotelefono)) {
43         return true;
44     } else {
45         return false;
46     }
47 }
```

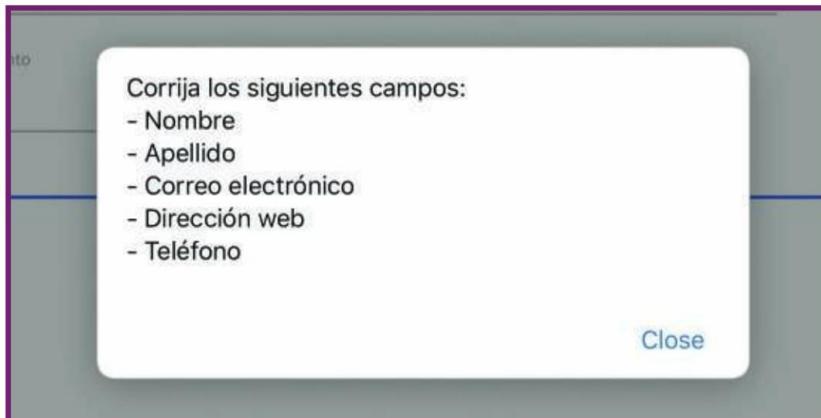
3 En la función que compara los valores ingresados en el campo Teléfono, utilizamos una expresión regular que solo acepte números del 0 al 9, y caracteres como espacio, guión, numeral, asterisco o paréntesis. Si comparamos esta expresión regular con el teclado numérico de un dispositivo móvil, veremos que todos estos caracteres pueden ser aceptados en el ingreso de un número telefónico.

```

49 function validoCampos() {
50     var esCorrecto = true;
51     var mErr = "Corrija los siguientes campos: \n";
52     if (!nombre_es_string()) {
53         mErr = mErr + "- Nombre \n";
54         esCorrecto = false;
55     }
56     if (!apellido_es_string()) {
57         mErr = mErr + "- Apellido \n";
58         esCorrecto = false;
59     }
60     if (!es_email()) {
61         mErr = mErr + "- Correo electrónico \n";
62         esCorrecto = false;
63     }
64     if (!es_url()) {
65         mErr = mErr + "- Dirección web \n";
66         esCorrecto = false;
67     }
68     if (!es_telefono()) {
69         mErr = mErr + "- Teléfono \n";
70         esCorrecto = false;
71     }
72     if (esCorrecto == false) {
73         alert(mErr);
74     } else {
75         alert('Los datos ingresados en el Formulario se han guardado');
76     }
77 }

```

4 Finalmente, creamos una función que realiza la invocación a las funciones anteriores, y en la devolución de TRUE o FALSE por parte de ellas, realiza una comparativa del resultado. Para aquellos casos en los que no se cumple la condición estipulada, agrega en la variable **mErr** el nombre del campo que hay que completar. Al final del algoritmo, si quedó uno o más campos sin completar, muestra al usuario un mensaje **Alert()** con los campos que debe completar o corregir. De lo contrario, graba o ejecuta el script de grabación de los datos del formulario y lo confirma mediante otro mensaje en pantalla.



5 Agregamos el llamado de la función **validoCampos()**, que compara el resto de las funciones, en el botón Enviar del formulario, y lo probamos sin agregar ningún valor en los campos. El resultado debe ser tal como se visualiza en la imagen. Ahora vayamos agregando valores al azar y pulsando el botón Enviar.

Expresiones regulares utilizadas

A continuación, un detalle de las expresiones regulares utilizadas, según el tipo de campo:

TIPO DE CAMPO	EXPRESIÓN REGULAR
URL	/^(http https):\/\/((\w+:{0,1}\w*)?(\S+)(:[0-9]+)?(\/ \/([\w#!:.?+=&%!\\-\\/])))?/
Email	/^[\w.-]+\@[^\w.-]+\.[^\w.-]{2,4}\$/.{1,}
Teléfono	/^[\+]*[(][0-9]{1,3}[)]{0,1}[-\s./0-9]*\$/g
Campos de texto	isNaN() significa NOT A NUMBER, y valida que el valor del campo no sea un valor numérico. No funciona como la opción más acertada, pero sí como la comparativa más rápida.

Solo nos queda pendiente resolver el **<Input>** tipo Select, que contiene valores en años. El lector puede animarse a resolverlo solo con lo aprendido hasta aquí.

Controlar el DOM HTML



En esta sección repasaremos los puntos importantes que brinda JavaScript para controlar el DOM (una API de programación para documentos) desde el mismo código de scripting, y así crear, modificar y eliminar cualquier componente HTML y sus clases CSS asociadas que necesitemos, de una manera fácil y dinámica.

Hasta ahora hemos manipulado las propiedades, métodos y comportamiento de los componentes de una página web a través de JavaScript, pero siempre creando previamente los componentes HTML desde el lenguaje propio de marcado. Veamos ahora

de qué modo podemos crear esos mismos componentes directamente desde JS, algo muy común en el estilo de desarrollo de aplicaciones web y móviles que proponen los diferentes frameworks JS actuales, como **Angular** y **React**, entre otros tantos.

Crear componentes HTML

A continuación, comenzaremos aprendiendo a crear componentes HTML desde nuestro código JS. Para hacerlo, descargamos desde el apartado de Material complementario de esta obra, el archivo **Fuentes-Ejercicio-08-base.zip**.

Extraemos los archivos que contiene (un HTML, un CSS y un JPG) y los agregamos a un proyecto nuevo en nuestro IDE Visual Studio. Veamos el código del archivo **index.html**:



```

1  <!DOCTYPE html>
2  <html>
3      <head>
4          <meta charset="utf-8" />
5          <meta name="viewport" content="width=device-width, initial-scale=1, maximum-scale=1">
6          <title>Creando componentes HTML</title>
7          <script src="index.js"></script>
8          <LINK href="hoja.css" rel="stylesheet" type="text/css">
9      </head>
10     <body>
11         <div id="campo" class="formato"></div>
12         <div id="boton" class="formato"></div>
13         <div id="imagen"></div>
14     </body>
15 </html>
```

Como podemos notar en la imagen, nuestro HTML tiene asociado un archivo CSS, un JS (que aún debemos crear) y, dentro del elemento **body**, solo tiene referenciado tres **DIVs**, cada uno de ellos con una clase CSS asociada. Creemos a continuación el archivo JS, presionando **CTRL + clic** sobre la referencia HTML a este archivo, y aceptando la sugerencia de VS Code de hacerlo.

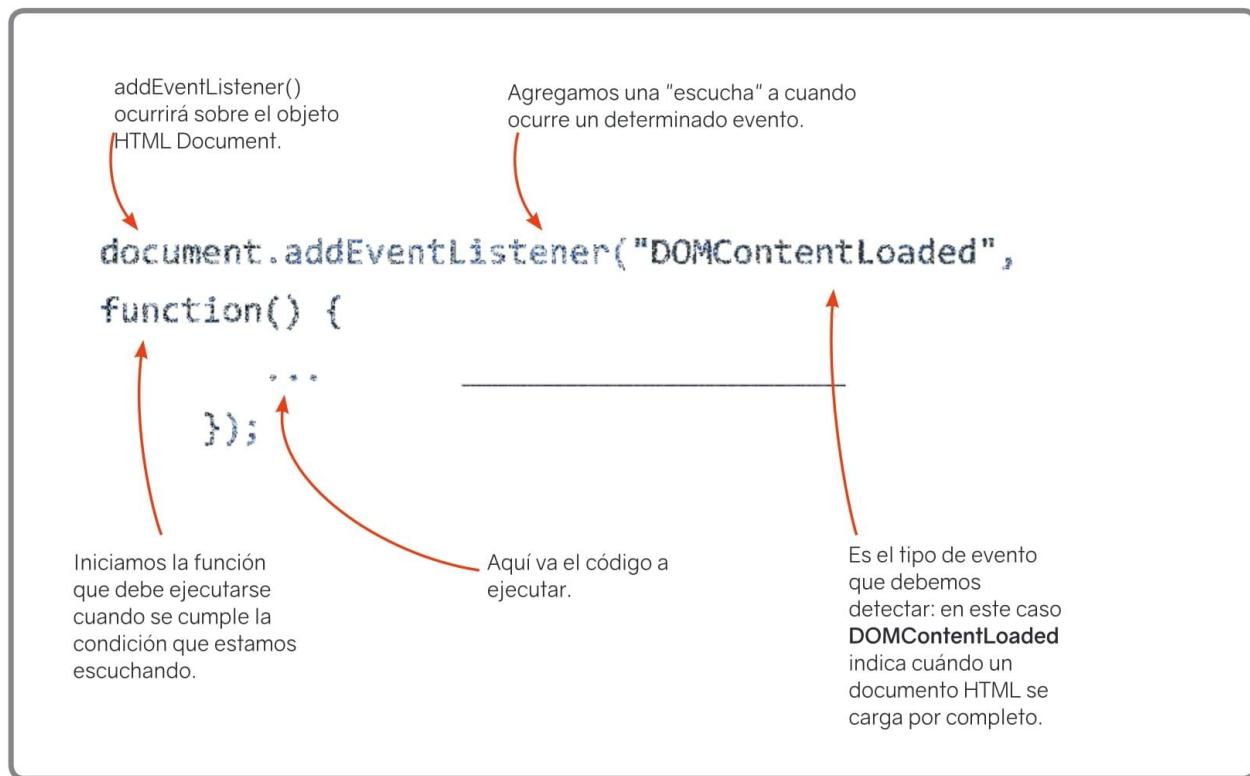
```

<title>Creando componentes HTML</title>
<script src="index.js"></script>
<LINK href="hoja.css" rel="stylesheet" type="text/css">
</head>
<body>
    <div id="campo" class="formato"></div>
    <div id="boton" class="formato"></div>
    <div id="imagen"></div>
</body>
</html>
```

Ya dentro del archivo HTML, la primera sentencia que escribimos es la siguiente:

```
document.addEventListener("DOMContentLoaded", function() {  
});
```

Analicemos a continuación el significado de este código:



CÓDIGO	DESCRIPCIÓN
document	Hace referencia al documento HTML en sí. Desde aquí podemos acceder a cada uno de sus componentes internos.
addEventListener	Esta función permite agregar un evento que se ocupa de validar, todo el tiempo, el comportamiento de un componente u objeto HTML. Cuando este cambia, ejecuta una función determinada (que debemos programar), que recibe dos parámetros.
"DOMContentLoaded"	DOMContentLoaded oficia como el primero de los parámetros de la función anterior. En este caso, lo que hace es estar atento al momento en que el documento HTML terminó de cargarse en el navegador web.
function()	Es la función propia que debemos escribir a continuación. El código dentro de esta función se ejecutará cuando el parámetro anterior cumpla su condición.

Agregar un campo de texto

Como vimos en ejercicios anteriores, los **INPUT TYPE text** nos permiten agregar campos de texto al documento HTML, a través de la sentencia:

```
<input type="Text" id="campodetexto">
```

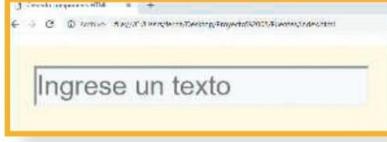
Veamos a continuación cómo hacer esto mismo, directamente desde el código JavaScript. Para lograrlo, y como parte de la función que aún no creamos para el código anterior, agregamos un componente **INPUT TYPE**:

La primera variable declarada 'd' toma la referencia al DIV cuyo ID es "campo".

La segunda variable declarada "t" engloba la creación del componente del tipo INPUT, utilizando la función `createElement()`.

A continuación, utilizamos el método `setAttribute()` para ir agregando las diferentes propiedades o atributos, con su respectivo valor.

Por último, mediante el método `appendChild()`, añadimos el componente creado dentro del DIV "campo". El resultado se aprecia en la siguiente imagen:



```

1  document.addEventListener("DOMContentLoaded", function() {
2      // Seleccionamos el DIV cuyo ID es 'campo'
3      var d = document.getElementById("campo");
4      // Creamos dentro de éste un INPUT TYPE TEXT
5      var t = document.createElement("INPUT");
6      t.setAttribute("type", "text");
7      t.setAttribute("value", "");
8      t.setAttribute("id", "micampodetexto");
9      t.setAttribute("placeholder", "Ingrese un texto");
10     d.appendChild(t);
11 }

```

En este ejemplo, le indicamos que el INPUT creado es del tipo TEXT, su contenido por defecto es vacío, su ID es **micampodetexto**, y el hint o **placeholder** que tendrá es "Ingrese un texto".

Como podemos ver, el INPUT TYPE es creado con todos los atributos visibles e invisibles que hemos declarado en el código JS. Y como el HTML tiene asociado un archivo CSS, el DIV (padre) donde creamos este INPUT TYPE mantiene sus características y colores declarados, como así también el INPUT TYPE hereda las especificaciones iniciales base que están declaradas en la hoja de estilos.

```

1 .formato {
2     background-color: #lightyellow;
3     padding: 15px;
4     border-width: 10px;
5     border-color: #gold;
6 }
7
8 button {
9     background-color: #brown;
10    color: #white;
11    font-size: 20px;
12    border-width: 1px;
13    border-color: #lightgreen;
14 }
15
16 input {
17     background-color: #ghostwhite;
18     font-size: 20px;
19 }

```

Crear un componente button

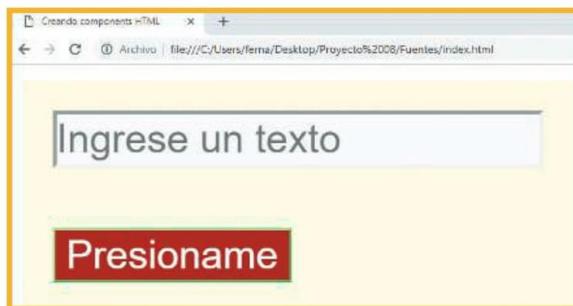
A continuación, crearemos un componente del tipo **button** desde el código JS.

Para hacerlo, luego del código del componente INPUT que creamos anteriormente, agregamos lo siguiente:

```
var d = document.getElementById("boton");
var b = document.createElement("BUTTON");
b.setAttribute("id", "btnPresionar");
b.innerText = "Presioname";
d.appendChild(b);
```

Como podemos ver en el código, repetimos la lógica utilizada en el INPUT TYPE, pero empleando los atributos propios del componente **button**: creamos una variable referenciando al elemento DIV (padre), creamos el componente HTML, establecemos sus propiedades y lo agregamos como elemento (hijo) al DIV.

El resultado se presenta en la siguiente imagen:



Si vemos otra vez el código de la hoja de estilo asociada al documento HTML, notaremos que los componentes del tipo **button** tendrán un estilo especial asociado. Es por esta razón que el botón se visualiza en color bordó con un borde verde y su texto en blanco.

Crear un componente image

Finalmente, agregaremos un componente del tipo imagen y adicionaremos a ella el archivo JPG que acompaña a este proyecto. Veamos el código a continuación:

```
var src = "logo-ru.jpg";
var i = document.getElementById("imagen");
var g = document.createElement("IMG");
g.setAttribute("src", src);
i.appendChild(g);
```

Aquí utilizamos la variable **src** para referenciar la imagen que mostraremos en el documento HTML. Luego capturamos el DIV imagen, creamos el elemento y agregamos el atributo **src** referenciando la variable homónima que contiene la imagen a visualizar. Por último, añadimos el componente creado al DIV correspondiente.

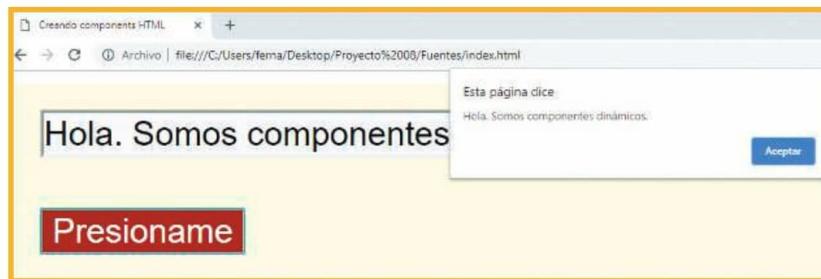


Capturar eventos de los componentes HTML

Con nuestros componentes HTML creados, veamos a continuación cómo agregarle comportamiento dinámico desde JS, sin modificar nada del documento HTML inicial. Para esto, en el archivo JS que venimos modificando, y siempre dentro de la función que creamos para detectar que el DOM se ha cargado totalmente, agregamos el siguiente código:

```
document.getElementById("btnPresionar").addEventListener("click", function() {  
    texto = document.getElementById("micampodetexto").value;  
    if (texto.trim() != "") {  
        alert(texto);  
    }  
});
```

¿Qué hace este código? ¡Fácil!, escucha cuando hacemos un clic sobre el botón, y entonces, ejecuta la función que verifica si hay un texto agregado en el componente **INPUT**. De haberlo, ejecuta la función **alert()** para mostrarlo.



Consideraciones de uso

Cuando construimos componentes o toda una interfaz HTML utilizando 100% código JS, debemos tener en cuenta una serie de aspectos para la construcción del código. Entre ellos, destacamos:

1 DOMContentLoaded

Siempre que nuestro archivo JS se cargue dentro del apartado head de un documento HTML, debemos recurrir a AddEventListener, y asegurarnos de que el HTML ya cargó antes de construir los componentes.

2 Estructurar la solución

Luego, debemos pensar siempre que dentro de la estructura del archivo JS hay que construir los componentes HTML primero, y luego ejecutar funciones JS que lean, escriban o los modifiquen.

3 Console.log()

Durante la construcción de la solución web, dentro de cada función JS, siempre conviene guiarnos implementando la función console.log(). De esta manera, detectaremos posibles fallas, y podremos eliminarlas o repensar nuestro código para que estas no afecten a la aplicación web.

Aplicar clases CSS a un componente

Si prestamos atención a cualquiera de las últimas imágenes, habremos notado que el componente HTML **imagen** contenido dentro de un DIV homónimo no tiene aplicado el mismo estilo que los DIVs anteriores. Veamos cómo manipular estos estilos desde JS:

```
var i = document.getElementById("imagen");
i.classList.add("formato");
```

Retornamos a nuestro documento JS, y en él ubicamos la declaración de la variable donde capturamos el elemento imagen (primera línea del código anterior). Luego, utilizamos esta misma variable para invocar la propiedad **classList**, que tiene asociado el método **.add()**. A este último método le pasamos como parámetro el nombre correspondiente a la clase que deseamos asociarle.



Para eliminar una clase asociada a un componente HTML, simplemente debemos realizar el mismo procedimiento anterior, pero cambiamos **.add()** por **.remove()**.

```
i.classList.remove(nombredelaclase);
```

Y si tenemos una clase determinada asociada a un componente HTML y queremos reemplazarla por otra clase, entonces debemos implementar el método **.replace()**.

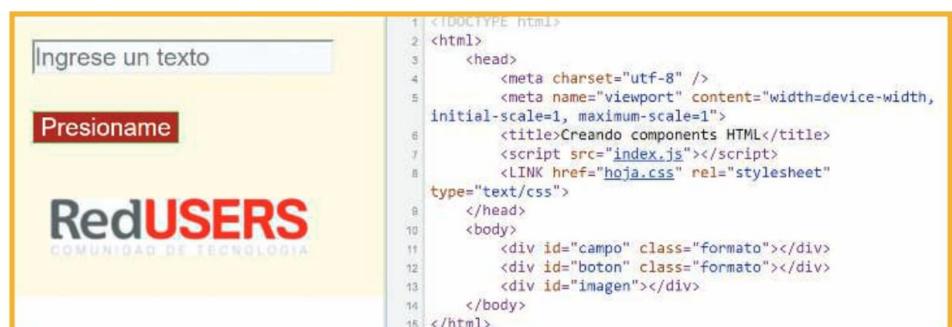
```
i.classList.replace("formato", "nuevoformato");
```

Frameworks modernos

El estilo de desarrollo de interfaces HTML partiendo de código JavaScript es el modelo implementado actualmente por la mayoría de los frameworks: Angular, React Native, Ionic, etcétera.

Por lo tanto, quienes piensen sumarse al desarrollo de aplicaciones web o web móviles, con las tecnologías en auge, tendrán que adoptar esta forma de desarrollo de soluciones HTML, partiendo desde JS.

A propósito, el código de nuestro documento HTML, luego de crear los componentes HTML desde JS, queda constituido como puede verse en la imagen.



Hacer hablar a JavaScript



En los últimos años, JavaScript pulió su capacidad de interpretar texto y convertirlo en audio a través de SpeechSynthesis. Veamos qué opciones nos ofrece el lenguaje en el campo del sintetizado de voces y cómo podemos expandir su capacidad hacia otros idiomas.

Debemos saber que esta característica aún se encuentra en modo experimental, por lo tanto, puede sufrir variaciones en la estructura del código u otros aspectos.

Aclarado este punto, veamos a través de la siguiente imagen qué navegadores web y cuáles sistemas operativos soportan actualmente esta capacidad.

	Desktop						Mobile					
	Chrome	Edge	Firefox	IE	Opera	Safari	Android	Chrome	Edge	Firefox	IE	Opera
Basic support	33	Yes	49	No	21	7	4.4.3	33	Yes	62	No	7.1
cancel	33	Yes	49	No	21	7	4.4.3	33	Yes	62	No	7.1
getVoices	33	Yes	49	No	21	7	4.4.3	33	Yes	62	No	7.1
onvoiceschanged	33	Yes	49	No	No	No	4.4.3	33	Yes	62	No	No
pause	33	Yes	49	No	21	7	4.4.3	33	Yes	62	No	7.1
paused	33	Yes	49	No	21	7	4.4.3	33	Yes	62	No	7.1
pending	33	Yes	49	No	21	7	4.4.3	33	Yes	62	No	7.1
resume	33	Yes	49	No	21	7	4.4.3	33	Yes	62	No	7.1
speak	33	Yes	49	No	21	7	4.4.3	33	Yes	62	No	7.1
speaking	33	Yes	49	No	21	7	4.4.3	33	Yes	62	No	7.1

Si prestamos atención a la imagen, veremos que se divide en dos tipos de plataformas: web de escritorio a la izquierda y dispositivos móviles a la derecha. A su vez, cada una de estas plataformas incluye los diferentes navegadores web habilitados para ellas. Como podemos apreciar, **Opera** es el único browser que por el momento soporta de manera limitada la característica de sintetizador de voz en su versión móvil, mientras

que **Microsoft Edge** directamente no la soporta. El resto de las plataformas, principalmente Chrome y Firefox, llevan muy bien la adopción del sintetizado de voz, tanto en desktop como en móviles.



ejemplo.html

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <title>JavaScript habla!</title>
5     <script>
6       function speak (message) {
7         var msg = new SpeechSynthesisUtterance(message);
8         var voices = window.speechSynthesis.getVoices();
9         msg.voice = voices[2];
10        msg.pitch = 0.5;
11        window.speechSynthesis.speak(msg);
12      }
13    </script>
14  </head>
15  <body>
16  </body>
```

Por lo tanto, con muy poco código, nuestro navegador web puede convertirse en un centro de información audible para los usuarios que navegan determinadas páginas web. Esta capacidad de JS permite extender fácilmente el soporte de sistemas basados en web para usuarios y centros donde se desea complementar las características

auditivas como soporte de una web o sistemas centralizados. Un ejemplo de esto pueden ser los sistemas de ordenamiento de personas, en los que cada vez que se invoca a alguien en una sala de espera, se muestre en un display el número que se llama y, a su vez, este dato sea leído de manera automática por la plataforma y reproducido por altoparlantes.

Ejercicio rápido

Creemos una página HTML llamada ejemplo.html, con la estructura básica, y copiemos dentro del bloque <HEAD> el siguiente código JavaScript:

```
function speak (mensaje) {
  var msg = new SpeechSynthesisUtterance(mensaje);
  var voices = window.speechSynthesis.getVoices();
  msg.voice = voices[2];
  msg.pitch = 0.5;
}
```

Recuerden que la función JS anterior debe ir dentro de los tags <script> y </script>. Y a continuación del cierre del tag </body>, el siguiente script:

```
var a = speak('Hola, mundo!');
```

Guardamos la página HTML y la ejecutamos en el navegador web. Como resultado, tendremos una página completamente en blanco, pero que, al finalizar su carga, reproduce el texto ingresado en la función **speak()**.

SpeechSynthesis

Analicemos a continuación el código escrito en el primer bloque:

Creamos una función denominada **Speak()**, la cual recibe un parámetro, en este caso, llamado mensaje.

La propiedad **voices[2]** indica que la sintetización de voz será en idioma **Español**, y la propiedad **Pitch** determina que la velocidad del audio será a la mitad de su valor predeterminado.

Luego creamos el objeto **msg**, tomando como base el tipo de objeto **SpeechSynthesisUtterance**. A este último, le pasamos el valor del parámetro **mensaje** (que reproducirá al ser invocado).

```
function speak(mensaje) {
    var msg = new SpeechSynthesisUtterance(mensaje);
    var voices = window.speechSynthesis.getVoices();
    msg.voice = voices[2];
    msg.pitch = 0.5;
    window.speechSynthesis.speak(msg);
}
```

Finalmente, reproducimos el contenido de texto almacenado en el objeto **msg**, tomando como base la configuración del resto de los parámetros que ajustamos.

El método **GetVoices()** nos permite obtener el listado de todos los idiomas soportados por el sintetizador de voz.

/<Ejercicio práctico>

Reproducir texto de un <input>

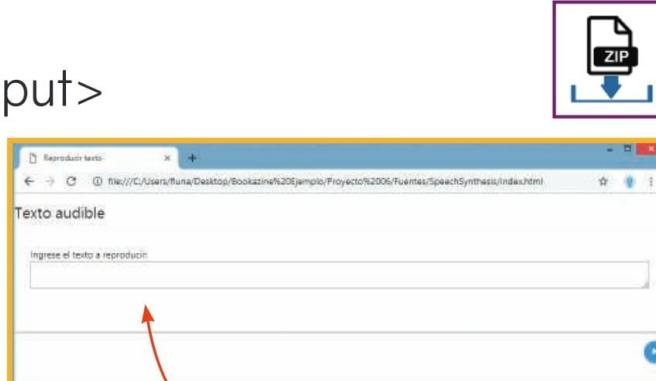
Realicemos a continuación un ejercicio para reproducir un texto ingresado mediante la función de sintetizador de voz de JavaScript. Descargamos de la web el ejercicio **Speech_Base.zip**, del material adicional que acompaña a esta obra. Generamos un nuevo proyecto en **VSCode**, agregamos el archivo **HTML** y creamos un archivo **JS** denominado **textoaudible.js**.

A continuación, abrimos el archivo **textoaudible.js** y escribimos en él la siguiente sintaxis:

Como vemos, creamos una variable denominada **texto**, la cual corresponde al objeto **SpeechSynthesisUtterance**.

Luego creamos una variable denominada **voces**, donde cargamos de forma dinámica todas las voces que el sintetizador de voz de JS soporta.

Acto seguido, creamos una función JS con el siguiente bloque de código:



El HTML descargado debe mostrarnos una web similar a la representada en la imagen.

```
var texto = new SpeechSynthesisUtterance();
var voces = speechSynthesis.getVoices();
texto.voice = '2';
```

Para terminar, establecemos la propiedad **voice** en **2**, que corresponde al idioma español.

```
function reproducirTexto() {
    var ta = document.getElementById('texto-audible').value;
    if (ta.length != '') {
        texto.text = ta;
        speechSynthesis.speak(texto);
    }
}
```

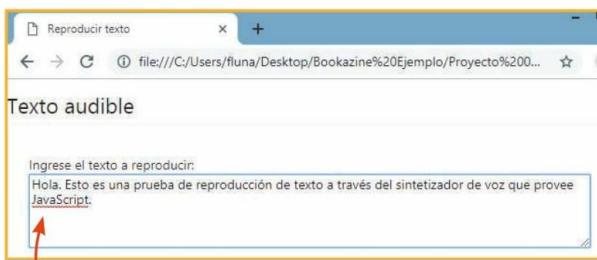
En la primera imagen vemos cómo queda estructurado nuestro código JS. Al analizarlo, podemos detectar que declaramos una variable **ta**, donde almacenamos el valor cargado en el cuadro de texto de la página HTML.

Luego utilizamos un bloque **IF** para determinar que la longitud del texto de esta variable tenga una longitud distinta de vacío, o sea, que contenga un texto. De ser así, entonces asignamos a la propiedad **text** del objeto **texto** el contenido de la caja de texto HTML.

```

1 var texto = new SpeechSynthesisUtterance();
2 var voces = speechSynthesis.getVoices();
3 texto.voice = '2';
4
5
6 function reproducirTexto() {
7     var ta = document.getElementById('texto-audible').value;
8     if (ta.length != '') {
9         texto.text = ta;
10    }
11    speechSynthesis.speak(texto);
12 }
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29

```



Agreguemos un texto escrito, en español, al cuadro de texto HTML para reproducirlo. Ya con la propiedad **text** asignada, pulsamos el botón Play para reproducir el objeto **texto** a través del método **speak()**, de la API **speechSynthesis**.

Calcular el tiempo de reproducción

Vamos a añadir unas líneas más de código en el archivo JS, para calcular el tiempo que tarda en reproducirse el audio que escribimos en la casilla. El código en cuestión para comenzar a medir el tiempo es el siguiente:

```

texto.onstart = function(e) {
    var t = event.elapsedTime / 1000;
    console.log('Tiempo de reproducción: ' + t.toFixed() + ' segundos.');
};

```

Y para finalizar la medición al momento en que termina la reproducción del sintetizador de audio, es este otro:

```

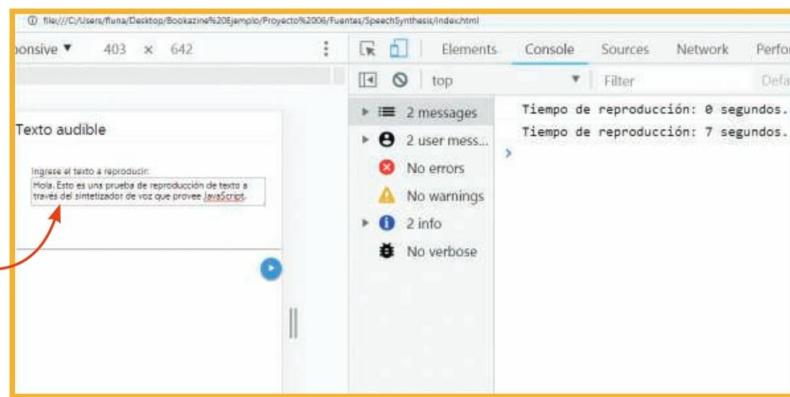
texto.onend = function(e) {
    var t = event.elapsedTime / 1000;
    console.log('Tiempo de reproducción: ' + t.toFixed() + ' segundos.');
};

```

Básicamente, **texto.onstart** almacena una función que comienza a medir el tiempo a través de la variable **t**, cuando pulsamos el botón reproducir.

A través de **texto.onend**, se almacena otra función que vuelve a medir el tiempo guardado en la variable **t**, para así estimar la demora de la reproducción, en segundos.

El resultado de esto debe ser similar al representado en la consola de depuración de la imagen. Mediante el método **toFixed()**, acotamos el tiempo en segundos a un número entero, y no con decimales.



Utilizar la API de notificaciones



Las notificaciones web, o notificaciones de escritorio, se han vuelto muy populares en este último año. Veamos qué nos propone JavaScript para que, a través de ellas, integremos fácilmente a nuestros desarrollos una interacción más fluida con el usuario.

Como usuarios que navegan constantemente la Web, habremos notado en este último tiempo que la mayoría de los sitios de **noticias, tecnología, e-commerce** y hasta las **redes sociales** nos ofrecen notificaciones de escritorio sobre los eventos y sucesos importantes que ocurren en ellos, o por parte de nuestros contactos en el caso de las redes sociales.

Estos mensajes que interactúan en el escritorio o pantalla del dispositivo móvil del usuario surgen para brindar información que las personas pueden catalogar como importante. Luego de



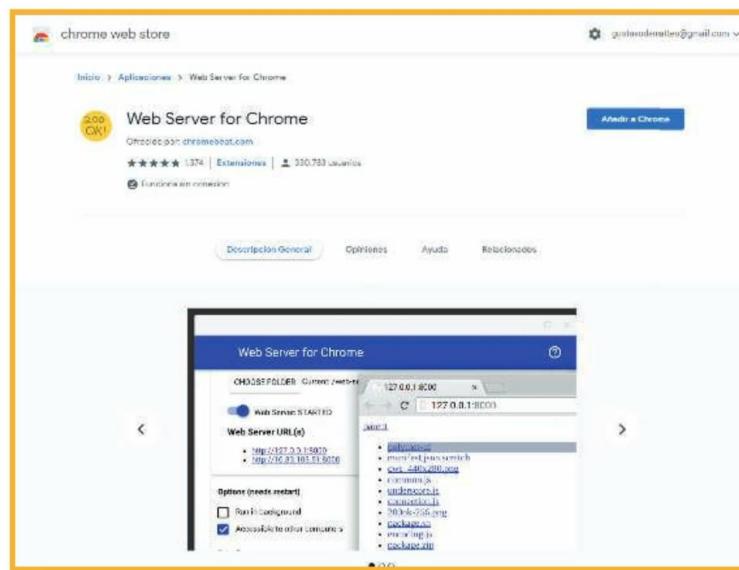
varias idas y vueltas en cuanto a la privacidad que cada uno desea o quiere tener, estas notificaciones deben realizarse solo a través del consentimiento

de los usuarios. Por lo tanto, el primer paso que debemos dar como desarrolladores es solicitarles permiso antes de emitir una notificación.

Preparación del ambiente de pruebas

A diferencia de otros ejercicios, el manejo de notificaciones requiere de la preparación previa de nuestro ambiente de pruebas. Para esto, debemos instalar un servidor web local que nos garantizará el correcto desarrollo de este proyecto. Por lo tanto, accedemos desde nuestro navegador web Chrome o Chromium al servicio de extensiones de Google Chrome y descargamos la extensión **Web Server for Chrome**: <http://bit.ly/2J0Frj3>. A continuación, pulsamos el botón **instalar**.

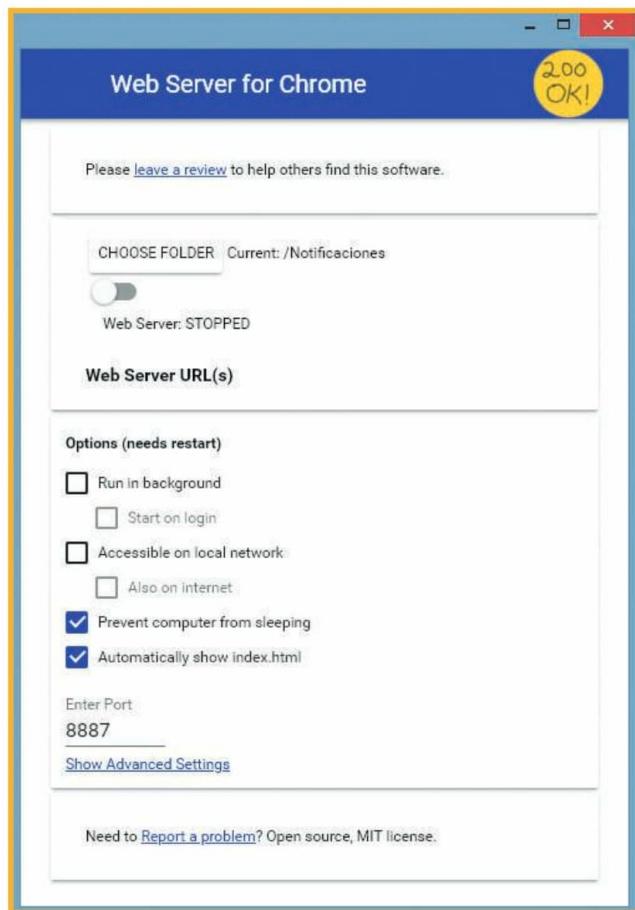
Finalizado el proceso, presionamos el botón **Iniciar Aplicación**, que abre la ventana de configuración y activación de esta extensión, similar a la que se ve en la imagen.



Configuración del servidor web

Este servidor web levanta en nuestra computadora el servicio necesario para emular un web server, mediante la URL **http://localhost:port** o **http://127.0.0.1:port**. En la siguiente infografía vemos los puntos clave de configuración rápida.

- 1 Pulsar el botón CHOOSE FOLDER.
- 2 Navegar hasta la carpeta del proyecto, seleccionarla y presionar ACEPTAR.
- 3 Cambiar el puerto o dejar por defecto la sugerencia.
- 4 Activar el switch para que el servidor comience a funcionar.



Finalmente, si tenemos un archivo denominado `index.html` en la carpeta que seleccionamos como origen de este servidor web, se mostrará accediendo a la URL correspondiente.



A continuación creamos una página web básica, denominada por supuesto **index.html**, y la guardamos en la carpeta que hemos configurado en el servidor web local. El código de la página web debe ser como el siguiente:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Notificaciones con JS</title>
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <script src="index.js"></script>
    <style>
      body { background-color: lightblue; }
    </style>
  </head>
  <body>
    <h3>Notificaciones con JS</h3>
    <button onclick="notificame()">Notificar!</button>
  </body>
</html>
```

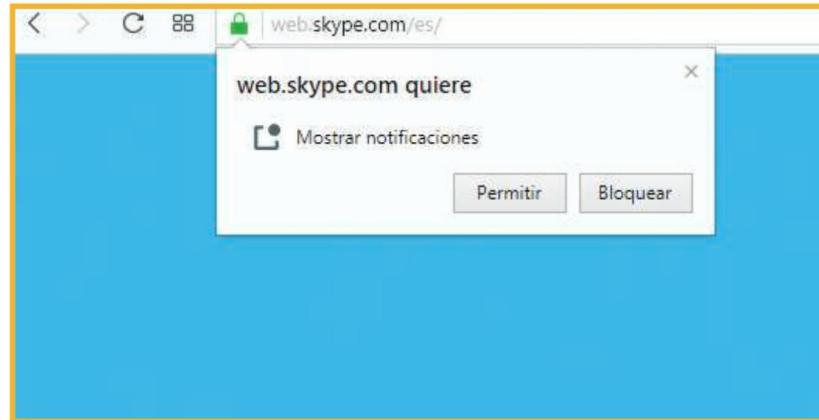
En este código incluimos un bloque de CSS para cambiar el fondo blanco predeterminado de la página web, solo para destacar la notificación que realizaremos a continuación.

Solicitar permisos al usuario

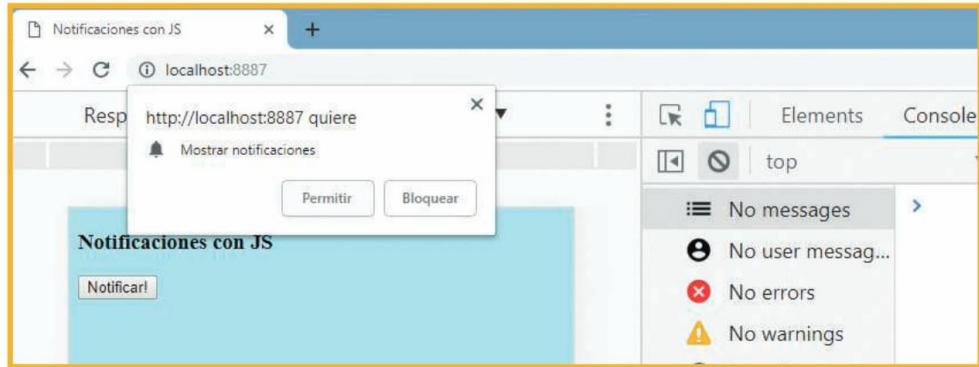
El primer paso para el uso de las notificaciones es solicitarle permiso de ejecución al usuario. Para hacerlo, JavaScript cuenta con una función específica. Creemos entonces el archivo JS que acompaña a nuestro **index.html**, y escribamos en él lo siguiente:

```
Notification.requestPermission().then(function(result) {
  console.log(result);
});
```

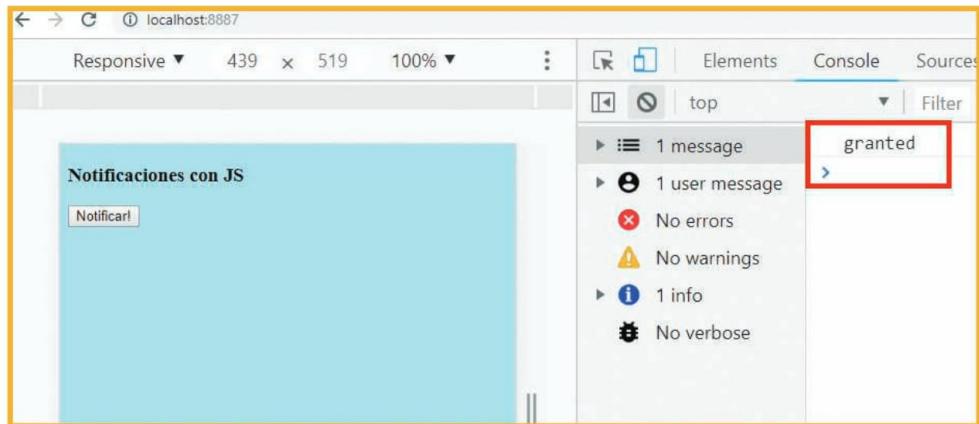
Lo primero que hacemos en este bloque de código es invocar de la API **Notification** su método **requestPermission()**. Este activa la ventana de solicitud de permiso propia de cada navegador.



Esta ventana modal le da al usuario dos opciones: **Permitir** o **Bloquear**. La primera permite ejecutar todas las notificaciones que nuestra web emita, mientras que la segunda las bloquea. La información de la decisión del usuario vendrá a nuestra aplicación web a través del parámetro **result** que vemos en el código. Utilizando **console.log(result)**, podremos observar la opción presionada.



Ejecutemos este ejemplo en el navegador web y, en la ventana modal, presionemos la opción **Permitir**. A través de la consola de Herramientas para el desarrollador, veremos la devolución de **result** elegida.



Continuamos ahora con la segunda parte del código, donde armamos la notificación en sí y la invocamos en pantalla; también prevenimos si el navegador es muy antiguo y no soporta notificaciones web.



Compatibilidad con navegadores web

Las notificaciones web llevan mucho tiempo funcionando en los diferentes navegadores de forma nativa, aunque debemos tener en cuenta que esta especificación JS aún se encuentra en su fase beta, por lo que puede sufrir cambios en

su estructura de código. Para conocer en detalle aquellos navegadores que la soportan y cuáles no, podemos consultar la ayuda oficial provista por **Mozilla Developer Network**, en el siguiente link: <https://mzl.la/2PCiOUA>.



The table provides a comprehensive overview of the Notification API's compatibility across different browser versions and platforms. Key findings include:

- Basic support:** Chrome (22), Edge (14), Firefox (22), Internet Explorer (No), Opera (25), Safari (6).
- Available in workers:** Chrome (45), Edge (Yes), Firefox (41), Internet Explorer (No), Opera (32), Safari (7).
- Secure contexts only:** Chrome (62), Edge (Yes), Firefox (No), Internet Explorer (No), Opera (49), Safari (7).
- Notification() constructor:** Chrome (22), Edge (Yes), Firefox (22), Internet Explorer (No), Opera (25), Safari (6).
- actions:** Chrome (53), Edge (18), Firefox (No), Internet Explorer (No), Opera (39), Safari (?).
- badge:** Chrome (53), Edge (18), Firefox (No), Internet Explorer (No), Opera (39), Safari (?).
- body:** Chrome (Yes), Edge (14), Firefox (Yes), Internet Explorer (No), Opera (?), Safari (?).
- data:** Chrome (Yes), Edge (16), Firefox (Yes), Internet Explorer (No), Opera (?), Safari (?).
- dir:** Chrome (Yes), Edge (14), Firefox (Yes), Internet Explorer (No), Opera (?), Safari (?).
- icon:** Chrome (22), Edge (14), Firefox (22), Internet Explorer (No), Opera (25), Safari (No).
- image:** Chrome (53), Edge (18), Firefox (No), Internet Explorer (No), Opera (40), Safari (?).
- lang:** Chrome (Yes), Edge (14), Firefox (Yes), Internet Explorer (No), Opera (?), Safari (?).
- maxActions:** Chrome (Yes), Edge (18), Firefox (No), Internet Explorer (No), Opera (?), Safari (?).
- onclick:** Chrome (Yes), Edge (14), Firefox (No), Internet Explorer (No), Opera (?), Safari (?).
- enclose:** Chrome (Yes), Edge (14), Firefox (Yes), Internet Explorer (No), Opera (?), Safari (?).
- onerror:** Chrome (Yes), Edge (14), Firefox (No), Internet Explorer (No), Opera (?), Safari (?).
- onshow:** Chrome (Yes), Edge (14), Firefox (Yes), Internet Explorer (No), Opera (?), Safari (?).
- permission:** Chrome (Yes), Edge (14), Firefox (Yes), Internet Explorer (No), Opera (?), Safari (?).
- renotify:** Chrome (50), Edge (No), Firefox (No), Internet Explorer (No), Opera (37), Safari (No).
- requireInteraction:** Chrome (Yes), Edge (17), Firefox (No), Internet Explorer (No), Opera (?), Safari (?).
- silent:** Chrome (43), Edge (17), Firefox (No), Internet Explorer (No), Opera (30), Safari (No).
- tag:** Chrome (Yes), Edge (14), Firefox (Yes), Internet Explorer (No), Opera (?), Safari (?).
- timestamp:** Chrome (Yes), Edge (17), Firefox (No), Internet Explorer (No), Opera (?), Safari (?).
- title:** Chrome (Yes), Edge (14), Firefox (No), Internet Explorer (No), Opera (?), Safari (?).
- vibrate:** Chrome (53), Edge (No), Firefox (No), Internet Explorer (No), Opera (39), Safari (?).
- close:** Chrome (Yes), Edge (14), Firefox (Yes), Internet Explorer (No), Opera (?), Safari (?).
- requestPermission:** Chrome (48), Edge (14), Firefox (47), Internet Explorer (No), Opera (40), Safari (?).

Legend:

- Full support (Green square)
- No support (Red square)
- Compatibility unknown (Grey square)
- * See implementation notes.
- Requires a vendor prefix or different name for use. (Note icon)

En la web Mozilla Developers encontramos un detalle de los parámetros, propiedades y métodos que componen la API Notification().

Para prevenir al usuario, o para saber como programadores si el navegador web no acepta notificaciones, podemos incluir el siguiente bloque de código:

```
if (!("Notification" in window)) {
    alert("La versión de su navegador web no soporta notificaciones web.");
    console.log('El navegador web no acepta notificaciones.');
}
```

Propiedades y métodos de la API Notification

Esta API cuenta con una serie de propiedades y métodos que nos conviene conocer para explotar mejor toda su funcionalidad. Veamos cuáles son en la siguiente tabla:

FUNCIONALIDAD	DESCRIPCIÓN
Actions	Permite mostrar una serie de acciones al usuario para que las ejecute, interactuando con la ventana Notification mostrada: <code>var actions[] = Notification.actions;</code>
Body	Representa el contenido en sí de la notificación; es decir, el texto que compone el mensaje visto por el usuario.
dir	Especifica la dirección del lenguaje que se utiliza para mostrar el texto en pantalla. Las opciones pueden ser Auto (la dirección por defecto del SO del usuario), ltr (de izquierda a derecha) o rtl (de derecha a izquierda).
icon	Acepta una URL absoluta en formato string, para mostrar un ícono que acompaña a la notificación en pantalla. Esta URL puede ser local o remota, aunque siempre nos conviene tener la imagen en forma local, para acortar tiempos de visualización de la notificación.
title	Se usa para especificar el título que contendrá la notificación en pantalla. A diferencia del resto de los parámetros, title se utiliza como una propiedad dentro del constructor de la instancia Notification.
lang	Permite especificar el lenguaje en el cual mostraremos el texto de la notificación. Por defecto se utiliza el establecido en el sistema operativo del usuario.
onclick	Permite establecer un Event Listener para escuchar el evento Clic sobre la ventana de notificación. Con esto, por ejemplo, podemos abrir un sitio web.
onclose	Al igual que onclick, establecemos un Event Listener que puede ejecutar un bloque de código, luego de que se cierra la ventana de notificación.
vibrate	En el caso de los dispositivos móviles que incluyen la opción de vibración, podemos ejecutar la función de vibrar, para que llame la atención del usuario.

Las notificaciones Web han comenzado a ser noticia por la excesiva cantidad de sitios web que las utilizan. Por ello, como desarrolladores debemos pensar muy bien su uso, para que no se tornen invasivas para el usuario final.



Construir la instancia Notification

Veamos entonces la estructura de la API, que debe estar instanciada en una variable JS, tal como se observa en el siguiente código:

```
var notificacion = new Notification(titulo, opciones);
```

Tal como se indica en la tabla anterior, **título** corresponde a la propiedad **title**, la cual se pasa como un parámetro.

Luego encontramos el siguiente parámetro, **opciones**, donde enviaremos el resto de las propiedades que terminan de dar forma a la ventana **Notification**.

Continuemos con el código JS. Debajo del bloque de código anterior, donde validamos que el navegador web soporte notificaciones, creamos la siguiente función:

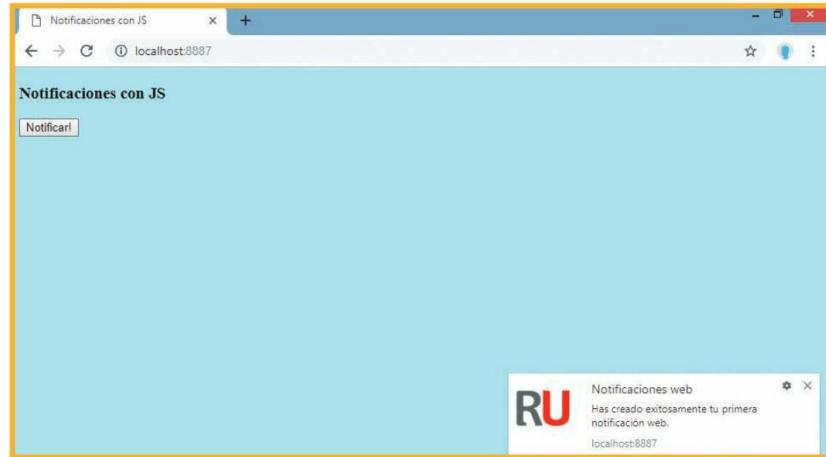
```
function notificame() {
    if (!("Notification" in window)) {
        alert("La versión de su navegador web no soporta notificaciones web.");
    }
    else if (Notification.permission === "granted")
        var imagen = '/images/logo-footer-transparente-128x128.png';
        var texto = 'Has creado exitosamente tu primera notificación web.';
        var titulo = 'Notificaciones web';
        var notificacion = new Notification(titulo, {
            body: texto,
            icon: imagen});
}
```

La estructura JavaScript final debe quedar tal como la muestra la siguiente imagen:



```
JS index.js x <> index.html
1  Notification.requestPermission().then(function(result) {
2      console.log(result);
3  });
4
5  function notificame() {
6      if (!("Notification" in window)) {
7          alert("La versión de su navegador web no soporta notificaciones web.");
8      }
9      else if (Notification.permission === "granted")
10         var imagen = '/images/logo-footer-transparente-128x128.png';
11         var texto = 'Has creado exitosamente tu primera notificación web.';
12         var titulo = 'Notificaciones web';
13         var notificacion = new Notification(titulo, {
14             body: texto,
15             icon: imagen});
16         console.log('Notificación emitida.');
17     }
18 }
```

En el bloque de código anterior, validamos que el usuario haya aceptado recibir notificaciones en su navegador. Luego creamos la variable **imagen**, donde establecemos un gráfico de **128x128 o 256x256** píxeles como ícono que se mostrará en la notificación. Seguido a esto, creamos la variable **texto**, donde escribimos el mensaje que se visualizará, el título que tendrá la notificación y, finalmente, la invocación a la misma. Solo nos queda



guardar todos los archivos de este proyecto y ejecutarlo en el navegador web para probar el resultado. Antes de la prueba,

verifiquemos que el botón creado con HTML contenga el llamado a la función JS **notificame()**.

Notificaciones temporales

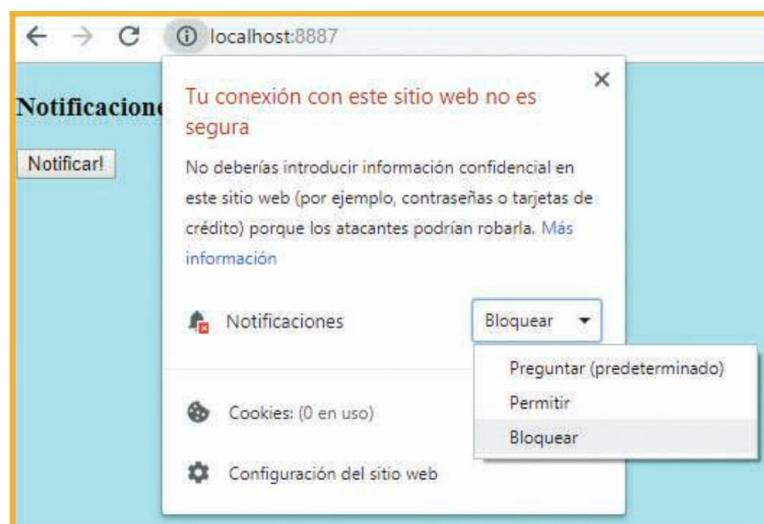
Si acorde al tipo de notificación que disparamos por código, queremos evitar que el usuario deba cerrarla para que desaparezca de la pantalla, podemos optar por utilizar la función JS **setTimeout()**, de modo que esta se cierre al cabo de unos segundos. El código para hacerlo es:

```
setTimeout(notificacion.close.bind(notificacion), 4000);
```

En este caso, la notificación se cerrará a los 4 segundos. Podemos cambiar el valor **4000** por cualquier otro, dentro de los parámetros **1000 a 60000** (desde 1 segundo como mínimo, hasta 60 segundos como máximo). Si no especificamos un tiempo determinado mediante **setTimeout()**, las notificaciones desaparecen de la pantalla a los 20 segundos (aproximados) de haberse ejecutado.

Probar el bloqueo

Para probar el funcionamiento de las notificaciones debemos cambiar el parámetro determinado de éstas. Realizamos esto pulsando el ícono **Información** o en la leyenda **No seguro**, ubicada a la izquierda de la URL del sitio web en cuestión, y seleccionando luego **Notificaciones/Bloquear**.



Es posible descargar el archivo Notificaciones Completo.zip, correspondiente a este proyecto, desde el sitio web de material adicional que acompaña a esta obra.

Almacenar datos localmente



Una de las premisas más importantes de la Web moderna es la capacidad de almacenar datos del lado del usuario. Es por eso que, en esta sección, veremos qué opciones de almacenamiento local tenemos y cómo trabajar con ellas, entre otras cosas, para reducir el consumo de datos.

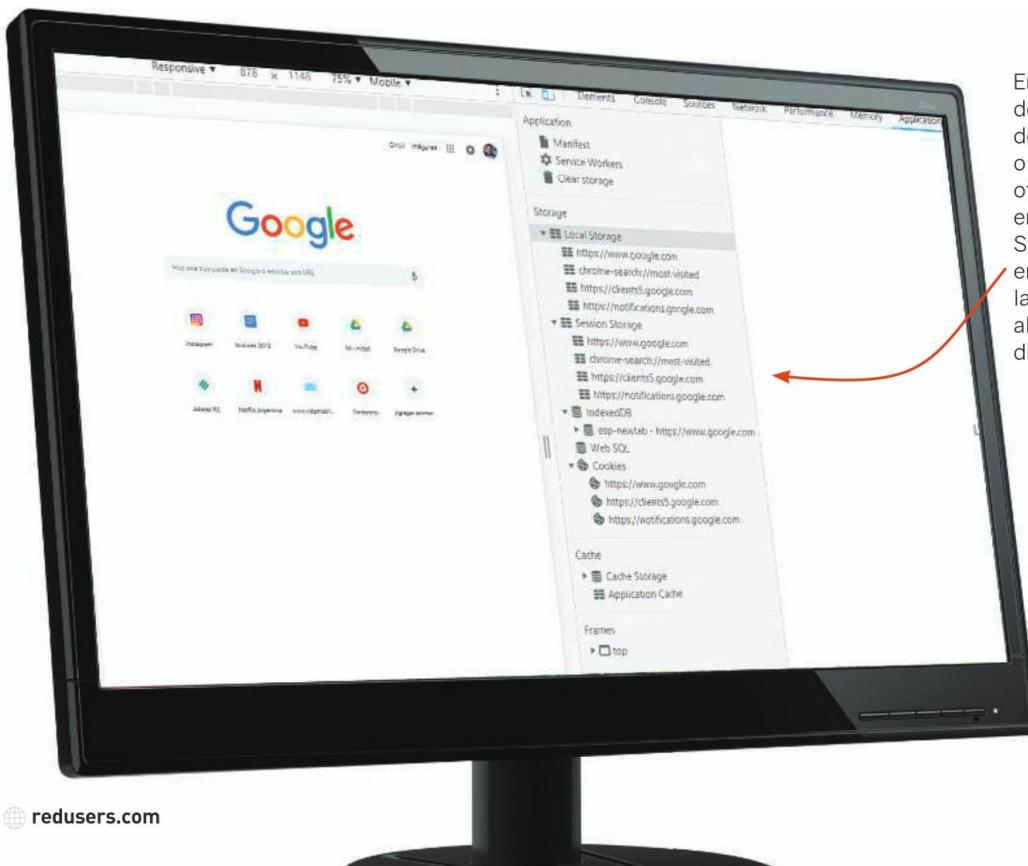
En los inicio de la Web, prácticamente la única forma de almacenar datos del lado del usuario era utilizando las conocidas **cookies**. Esta opción permitía, por ejemplo, conocer cuántas veces un usuario nos había visitado, guardar su nombre u otro dato de importancia, y recuperarlo cuando volvía a visitar la web. Las cookies nos ofrecían, entre otras cosas, el poder contar con soporte heredado, hacer que los datos fueran persistentes y poner una fecha de caducidad.

Como desventaja, cada dominio almacena sus propias cookies en una única cadena, lo cual complica la posibilidad de analizar claramente los datos guardados. Estos tampoco se encriptan, a menos que lo hagamos nosotros mediante un algoritmo propio. Su capacidad de almacenamiento se limita a 4 Kilobytes, y lo peor de todo es que **SQL INJECTION** puede hacerse tranquilamente desde ellas a una BBDD en la nube.

La nueva era de almacenamiento web local

Entre las opciones que nos ofrece la modernización de los navegadores web, gracias a la llegada de HTML5 y su complemento perfecto con JavaScript, podemos encontrar las siguientes:

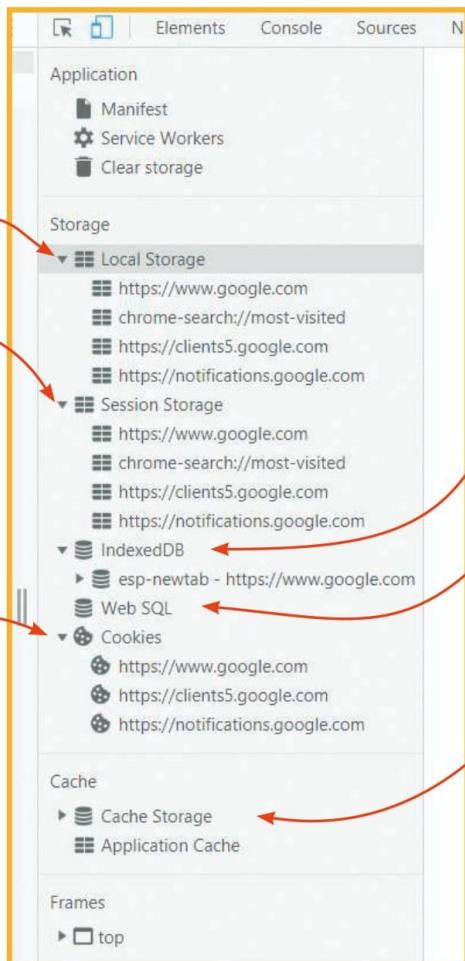
- LocalStorage
- SessionStorage
- IndexedDB
- Web SQL
- Application Cache



En Herramientas de desarrollador de Google Chrome o cualquier otro browser, en Application/Storage encontramos la oferta de almacenamiento disponible.

Opciones de almacenamiento del lado del usuario

A través de la siguiente infografía, veamos una breve descripción de las diferentes opciones que tenemos hoy para guardar datos en forma local y, así, agilizar los tiempos de carga o recuperación de la información:



The screenshot shows the Chrome DevTools Application tab with the Storage panel open. The Storage panel lists several categories of local storage:

- LocalStorage**: Contains items like https://www.google.com, chrome-search://most-visited, etc.
- Session Storage**: Contains items like https://www.google.com, chrome-search://most-visited, etc.
- IndexedDB**: Contains a single item: esp-newtab - https://www.google.com
- Web SQL**: Contains a single item: esp-newtab - https://www.google.com
- Cookies**: Contains items like https://www.google.com, https://clients5.google.com, https://notifications.google.com
- Cache**: Contains **Cache Storage** and **Application Cache**.

Red arrows from the text descriptions point to their corresponding sections in the DevTools interface.

LocalStorage
Es la opción más implementada, que prácticamente reemplazó a las clásicas cookies, y es la que veremos a lo largo de esta sección.

SessionStorage
Se implementa principalmente para iniciar y mantener una sesión de datos temporales, los cuales garantizan que, al cerrar la pestaña o el navegador web, estos sean eliminados del disco local.

Cookies
Por tradición, compatibilidad con aquellas web que aún utilizan tecnologías antiguas, y para mantener una segunda opción ante la ausencia de LocalStorage, las cookies siguen teniendo su lugar en el ecosistema de almacenamiento del lado del usuario.

IndexedDB
Permite crear una base de datos local y transaccional, utilizando la estructura de objetos del tipo JSon (par - valor), con índices.

Web SQL
Es una API web que permite almacenar datos localmente, utilizando el lenguaje de datos SQL para cada consulta, grabación y eliminación. Solo es compatible con los navegadores Chrome, Opera, Safari y Android Browser.

Cache Storage
Esta opción se usa solo para almacenar páginas HTML, documentos CSS y archivos JS del lado del usuario, y también archivos JSon, imágenes, videos e iconos. Es utilizada siempre en las implementaciones basadas en PWA.

La mayoría de las ofertas de almacenamiento local va cambiando con el tiempo. Lo más conveniente es que elijamos una opción de almacenamiento local que sea funcional y común a todos los navegadores web.



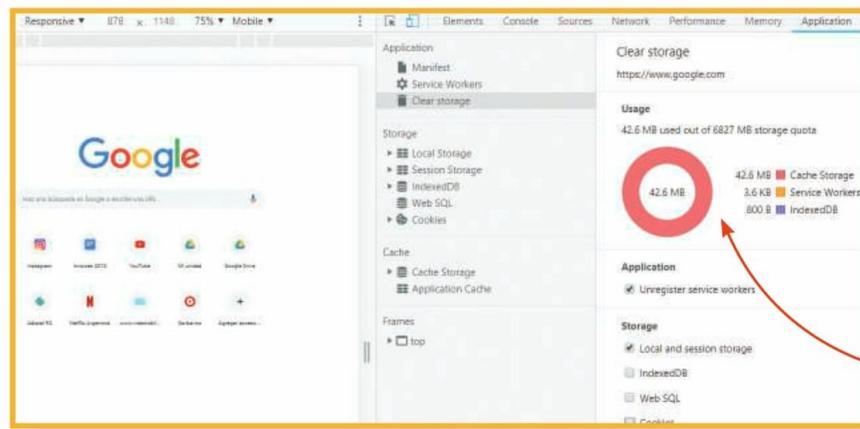
LocalStorage

Veamos a continuación cuáles son las características más importantes de LocalStorage y cómo trabajar de manera eficaz con esta opción de almacenamiento de datos local.

Para reemplazar rápidamente a las cookies, utilizando a su vez una forma óptima de leer y almacenar datos, LocalStorage es la opción que necesitamos implementar.

Entre sus características más beneficiosas, podemos destacar:

- Soporta los navegadores modernos.
- Almacena directamente en el browser y no en un archivo local.
- No se envían los datos almacenados con cada petición http.
- Posee 5120 KB (o 5 MB) de almacenamiento por cada dominio.



Como podemos ver, las webs modernas que aprovechan la capacidad de almacenamiento local terminan consumiendo un espacio importante del lado de la máquina cliente (nuestro navegador).

Sus métodos

LocalStorage cuenta con una serie de métodos muy simples que le permiten realizar las operaciones básicas con los datos que se almacenan del lado del usuario. Veamos en la siguiente tabla cuáles son estos métodos:

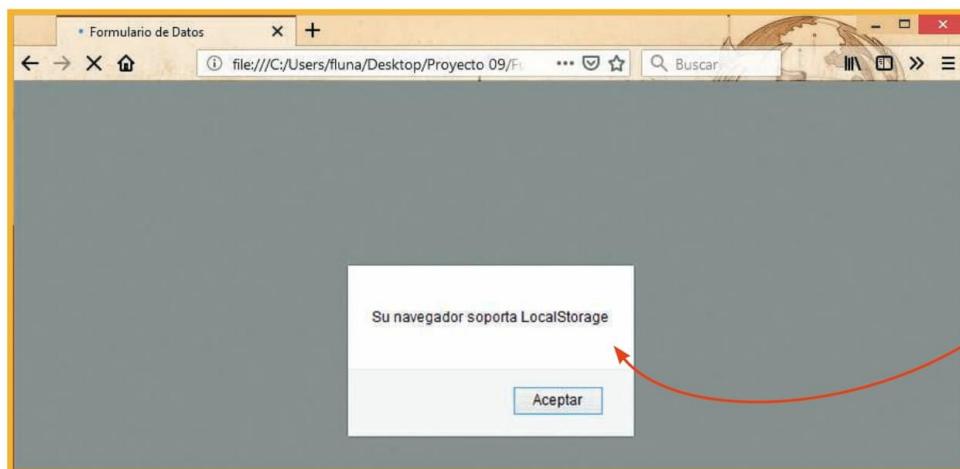
MÉTODO	DESCRIPCIÓN
Length()	A través de este método podemos conocer la longitud del total de datos almacenados del lado del usuario. Es útil para saber qué capacidad de los 5 MB disponibles hemos consumido hasta el momento.
setItem(a, b)	Este método recibe dos parámetros: (a) indica el nombre con el cual identificaremos el dato por guardar, y (b) es el valor que deseamos almacenar dentro de ese dato.
getItem()	Permite obtener el valor almacenado en un determinado ítem existente. Este ítem debe pasarse al método como parámetro, y almacenar su resultado en una variable o directamente en un componente web.
removeItem()	A través de este método, podemos eliminar un ítem existente junto con su valor almacenado. Recibe como parámetro el ítem en cuestión que deseamos borrar del almacenamiento local.

Todos estos métodos se aplican sin modificación alguna sobre **SessionStorage**, la variante que vimos para almacenar datos localmente, solo que de forma temporal. Veamos ahora la manera de implementar exitosamente estos métodos para sacar provecho de las capacidades de **LocalStorage**.

Soporte para almacenamiento local

Si queremos asegurarnos de que el navegador web que carga el sitio soporta **LocalStorage**, entonces podemos implementar, al inicio del HTML inicial, un código similar al siguiente:

```
<script>
  if (localStorage) {
    alert("Su navegador soporta LocalStorage");
  } else {
    alert("Actualice su navegador web para continuar navegando
en este sitio.");
  }
</script>
```



Si bien es un poco rudimentario preguntar hoy si un navegador web soporta LocalStorage, de esta manera sabremos con certeza que el web browser que está cargando nuestro sitio funcionará de forma óptima con Local y Session Storage.

LocalStorage.setItem()

La sentencia para almacenar un dato del lado del usuario es:

```
localStorage.setItem("colorDeFondo", "add8e6"); //celeste
```

Esta opción podría aplicarse muy bien para guardar del lado del usuario el color de fondo de nuestro sitio web.

LocalStorage.getItem()

Continuando con el ejemplo anterior, ahora debemos recuperar el color de fondo guardado, para mostrarlo en el documento HTML en cuestión.

```
Var cdf = localStorage.getItem("colorDeFondo");
document.body.style.backgroundColor = cdf;
```

LocalStorage.removeItem()

Y si no deseamos almacenar más el dato, simplemente invocamos la sentencia:

```
localStorage.removeItem("colorDeFondo");
```

Y así, el dato almacenado será eliminado del navegador web del usuario.

/<Ejercicio práctico>

Salvar datos de manera progresiva

Luego de este repaso rápido, implementemos una solución útil que demuestre dónde podría aplicarse muy bien el almacenamiento y la recuperación de datos de manera local. Para este ejemplo, vamos a recuperar primero el ejercicio del Formulario de datos que realizamos anteriormente. Abrimos con el editor el archivo index.html, y a continuación de la declaración `<script src="index.js">...`, agregamos la siguiente sentencia:

```
<script src="formulario.js" defer></script>
```

¿Qué característica nueva implementamos en la declaración de este script? El uso del atributo **Defer**. Pero ¿para qué sirve? Veámoslo a continuación.



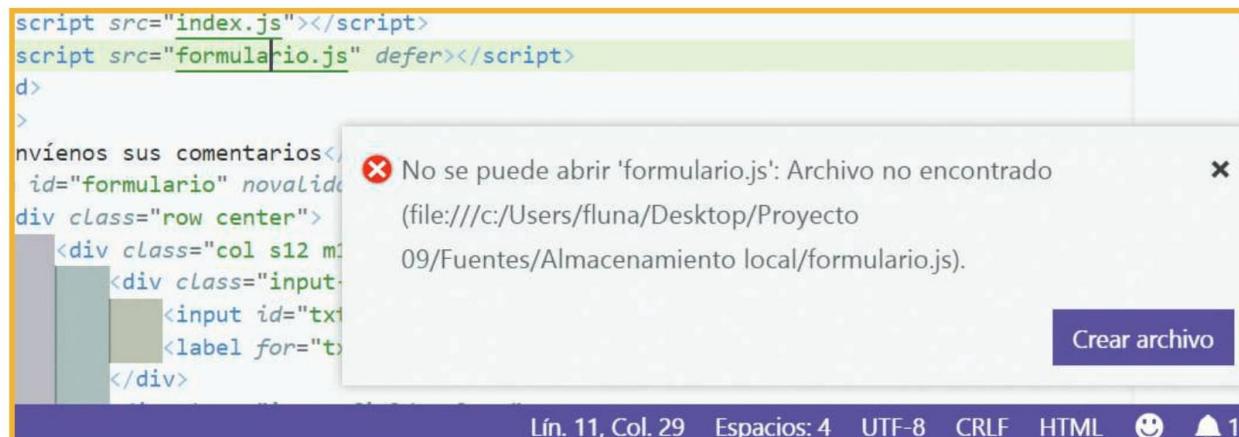
```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Formulario de Datos</title>
    <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/mater...
    <!-- Aquí Referencia a INDEX.JS -->
    <script src="index.js"></script>
    <script src="formulario.js" defer></script>
  </head>
  <body>
    <h5>Envíenos sus comentarios</h5>
    <form id="formulario" novalidate>
      <div class="row center">
```

Atributo defer

Al implementar el atributo **defer** con la referencia al archivo JavaScript, la carga de este JS del lado del usuario se realizará una vez finalizada la carga del documento HTML y sus referencias. Así, evitamos que nuestro código JS falle en su ejecución, referenciando a un componente HTML no creado al momento de su renderizado.

Crear dinámicamente el archivo JS

Pulsamos la tecla CTRL seguida de un clic sobre el archivo **formulario.js**.



```
script src="index.js"></script>
script src="formulario.js" defer></script>
d>
>
nviénenos sus comentarios</h5>
<form id="formulario" novalidate>
  <div class="row center">
    <div class="col s12 m12 l12 xl12">
      <div class="input-field">
        <input id="txtComentario" type="text" />
        <label for="txtComentario">Comentario</label>
      </div>
    </div>
```

No se puede abrir 'formulario.js': Archivo no encontrado
 (file:///c:/Users/fluna/Desktop/Proyecto 09/Fuentes/Almacenamiento local/formulario.js).

Crear archivo

Lín. 11, Col. 29 Espacios: 4 UTF-8 CRLF HTML

Ya creado el archivo JS, le agregamos a continuación el siguiente código funcional:

Escuchamos el evento **DomContentLoaded** del documento HTML. Una vez cargado todo el documento HTML, recién entonces procesaremos el código JS siguiente:

El método **QuerySelectorAll()** permite referenciar a una serie de componentes HTML en modo array, los cuales serán "escuchados" para luego ejecutar un evento determinado. Acto seguido, recurrimos a la sentencia JS **For**, la cual recorrerá estos componentes HTML para detectar cuando un evento determinado ocurra.

Esta sentencia analiza cada elemento, denominado **valor**, dentro de la colección o array de elementos contenido en la variable **valores**. A cada elemento se le asigna un evento "**change**", mediante el método **AddEventListener**, que detectará un cambio en él:

this.id toma como nombre el id del componente HTML, y **this.value** guarda el valor escrito dentro del componente (en este caso, INPUT).

Aprovechando el **For** y la recorrida por cada uno de los elementos almacenados en la colección **valores**, creamos una variable **campoGuardado**, donde almacenamos el dato guardado localmente. Si dicha variable no está vacía, entonces mostramos en el componente HTML el valor recuperado del almacenamiento local.

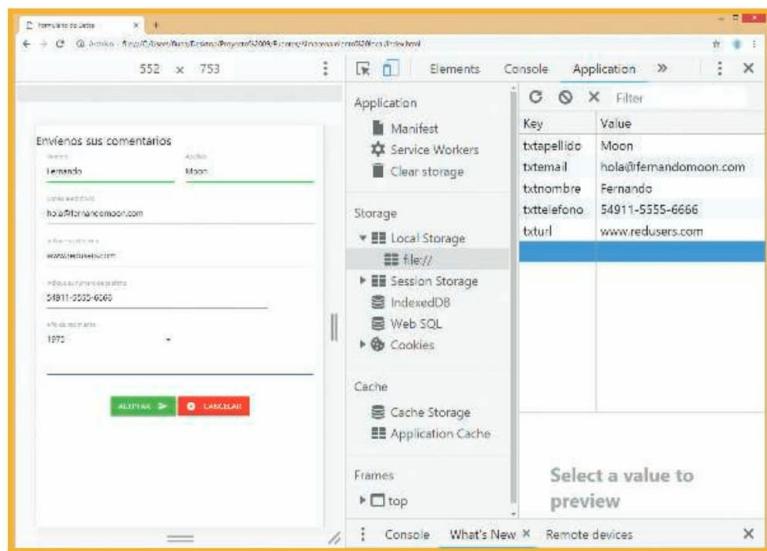
```
document.addEventListener("DOMContentLoaded",
  function(){
});
```

```
var valores = document.querySelectorAll("#txtnombre,
#txtapellido, #txtemail, #txturl, #txttelefono");
```

```
for (var valor of valores) {
  ...
}
```

```
...
//Guardamos el valor de cada campo
valor.addEventListener("change", function() {
  localStorage.setItem(valor.id, valor.value);
});
...
```

```
...
//Recuperamos el valor de cada campo guardado
var campoGuardado = localStorage.getItem(valor.id);
if (campoGuardado != null) {
  valor.value = campoGuardado.trim();
}
...
```



Finalmente, podemos ver a través de la imagen que nuestro ejemplo de almacenamiento local utilizando **LocalStorage** funciona de manera efectiva. Si cerramos el documento HTML o recargamos la página correspondiente, veremos que los datos son recuperados del almacenamiento local y visualizados correctamente en cada uno de los campos de dicho formulario. Este es un buen y efectivo ejemplo que puede ayudar a autocompletar un formulario o a recuperar datos ya cargados ante un cuelgue del navegador web.

Implementar SQL offline



El lenguaje de consulta estructurado, conocido por todos como SQL, tiene varias décadas de maduración, y es por eso que también tiene su lugar en el mundo del almacenamiento de datos offline, desde una aplicación web.

Veamos entonces cómo implementarlo exitosamente en nuestros proyectos web.

Si bien Local y Session Storage son dos herramientas fáciles de implementar al momento de tener que almacenar datos localmente, cuando llegamos a cierto punto de complejidad en el desarrollo de una alternativa web, debemos pensar en una solución un poco más efectiva.

Y si somos programadores con experiencia

o recién estamos incursionando en el mundo del desarrollo web, el manejo del **lenguaje SQL (Structured Query Language)** es la opción más efectiva cuando debemos manipular un compendio de datos de mediano a grande. Veamos entonces cómo aprovechar las características básicas pero efectivas del lenguaje SQL dentro de nuestras soluciones web.

De qué se trata WebSQL

Web SQL es una adaptación que funciona comúnmente en los navegadores que utilizan el motor **WebKit** o **Blink** (Chrome, Chromium, Opera y Safari), y que, a través de simples métodos, se emplea para crear una base de datos relacional del lado del usuario. Dicha base se almacena dentro de una instancia que permite el uso exclusivo de la aplicación web que la creó. Web SQL utiliza tres métodos simples:

- **Opendatabase** se usa para abrir o crear una base de datos. Recibe una serie de parámetros elementales, los cuales le permitirán ejecutarse de manera eficaz.
- **Transaction** es el método utilizado para realizar operaciones de escritura sobre la base de datos y/o tablas que esta contiene.
- **ExecuteSql** realiza la lectura de los datos ya creados y almacenados localmente en tablas.

Introducción a Web SQL

Archivo | file:///C:/Users/fiuna/Desktop/Proyecto%2010/Fuentes/Web%20SQL/index.html

Responsive ▾ 507 x 840 100% ▾ Mobile ▾ Online ▾

Elements Console Sources Network

Base de datos Web SQL

Creación de nuestra primera base de datos SQL del lado del usuario.

Application
Manifest
Service Workers
Clear storage

Storage
Local Storage
Session Storage
IndexedDB
Web SQL
Cookies

Cache
Cache Storage

Dentro de las herramientas para el desarrollador del navegador Chrome, o en cualquiera de los otros antes mencionados, podemos encontrar el apartado **Application > Web SQL**, donde se lista la o las bases de datos que cree nuestra solución web.

Crear una base de datos

Iniciemos un nuevo ejercicio con un documento HTML y un archivo JS asociado, donde realizaremos todos los pasos que veremos a continuación. Dentro del archivo JS, escribimos el siguiente código:

```
document.addEventListener("DOMContentLoaded", function() {
    var db = openDatabase("tareas", "1.0", "mi base de datos de tareas", 2
* 1024 * 1024);
    ...
});
```

Como podemos ver, esperamos a que se cargue el documento HTML y luego creamos la base de datos en cuestión. Si ésta existe, el método la abrirá, y si no, la creará desde cero.

Crear una tabla

Para crear una tabla dentro de la misma base de datos, primero debemos abrir una transacción a través del método **transaction()**, y luego, invocar el método **executeSql()**.

```
...
db.transaction(function (tx) {
    tx.executeSql('CREATE TABLE mistareas (id unique, text)');
});
```

¿Cómo resolvemos la creación de una tabla si esta ya existía antes?

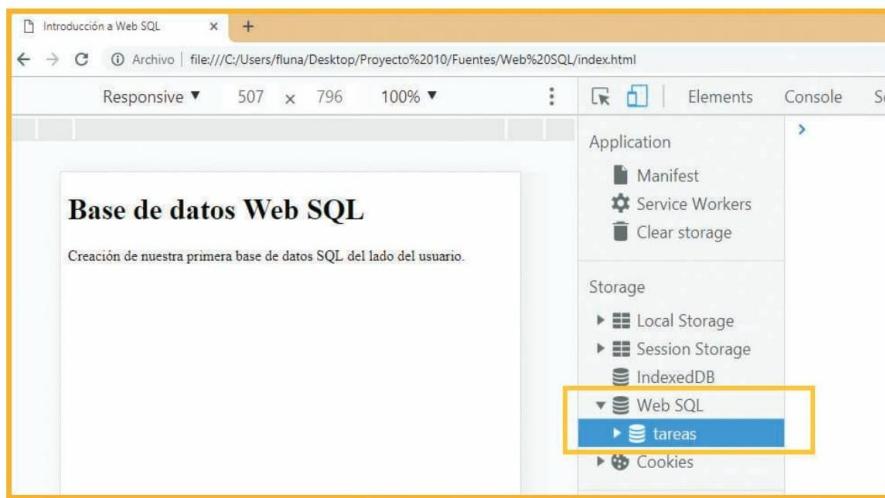
De la misma manera que lo hacemos en SQL: integramos en la sentencia **CREATE TABLE** la condición **IF NOT EXISTS** antes del nombre de la tabla, tal como vemos en el siguiente código:

```
tr.executeSql('CREATE TABLE IF NOT EXISTS mistareas (id unique, text)');
```

The screenshot shows the Chrome DevTools Storage panel. On the left, there's a large orange-bordered area labeled "Base de datos Web SQL" with the sub-instruction "Creación de nuestra primera base de datos SQL del lado del usuario.". On the right, the Storage panel lists various storage types: Manifest, Service Workers, Clear storage, Storage (Local Storage, Session Storage, IndexedDB), and Cookies. Under Storage, the IndexedDB section is expanded, showing a tree structure for "Web SQL". A specific database named "tareas" is selected, and under it, a table named "mistareas" is visible. The entire "Web SQL" section is highlighted with a yellow border.

Parámetros del método openDatabase()

Este método recibe cuatro parámetros: el nombre de la base, la versión de la base, una breve descripción y el tamaño inicial. Cualquiera de estos parámetros que obviemos en el llamado del método **openDatabase()** impedirá la creación o apertura de la base de datos.



Agregar registros a la tabla

Veamos ahora cómo se estructura la sentencia que permite agregar registros a una tabla:

```
tr.executeSql('INSERT INTO mistareas (id, text) VALUES (1, "Así se visualiza una tarea");  
tr.executeSql('INSERT INTO mistareas (id, text) VALUES (2, "Esta es otra tarea");  
tr.executeSql('INSERT INTO mistareas (id, text) VALUES (3, "Y aquí, una tercer tarea");
```

	rowid	id	text
	1		Así se visualiza una tarea

Consideraciones con el índice

Como vimos en el ejemplo anterior donde insertamos registros, estamos poniendo manualmente el índice o ID, que corresponde a cada uno de los registros insertados. Cuando creamos la tabla, declaramos el campo ID como índice único, por lo que no se pueden duplicar los registros insertados. Web SQL directamente obvia la inserción de ID duplicados evitando la

inserción del registro, pero a su vez, no devuelve un mensaje ni inserta el registro nuevo.

Para solucionar este problema, debemos ejecutar una sentencia SQL que obtenga primero el último ID que se ha insertado y, luego, le sume uno para poder agregar efectivamente un nuevo registro con el ID consecutivo.

¿Cómo lo conseguimos?:

```
tr.executeSql('SELECT * FROM mistareas ORDER BY id DESC LIMIT 1', [], function(tr, results) {
    var len = results.rows.length, i;
    if (len == '') { //La tabla MISTAREAS está vacía
        id = 1;
    } else { //Existen registros en la tabla
        id = results.rows.item(i).id;
        id++;
    }
    tr.executeSql('INSERT INTO mistareas (id, text) VALUES (' + id + ', "Así se visualiza una tarea")');
```

Si analizamos el código, lo primero que hacemos es leer el campo **ID** de la tabla **mistareas**, ordenando de manera descendente el resultado, y filtrando mediante **LIMIT 1**, para listar un único registro.

En la variable **len**, guardamos la longitud del resultado obtenido. Si no existen registros, la variable **len** quedará vacía. Luego preguntamos si esta variable está vacía, y en ese caso, asignamos a la variable **ID** el valor **1**. Si no está vacía, obtenemos el ID devuelto, lo guardamos en la variable **ID** y le sumamos **1**. Finalmente, ejecutamos la sentencia **INSERT**, reemplazando el número fijo del ID por la variable en cuestión.

Refresquemos varias veces el documento HTML, para generar sendos registros.

The screenshot shows the Chrome DevTools Application tab with a list of storage types: Session Storage, IndexedDB, Web SQL, Cookies, Cache, Cache Storage, Application Cache, and Frames. Under the Web SQL section, there is a table named 'tareas' with a single object named 'mistareas'. The table has columns: rowid, id, and text. The data is as follows:

	rowid	id	text
Session Storage	1	1	Así se visualiza una tarea
IndexedDB	2	2	Así se visualiza una tarea
Web SQL	3	3	Así se visualiza una tarea
tareas	4	4	Así se visualiza una tarea
mistareas	5	5	Así se visualiza una tarea
Cookies	6	6	Así se visualiza una tarea
Cache	7	7	Así se visualiza una tarea
Cache Storage	8	8	Así se visualiza una tarea
Application Cache	9	9	Así se visualiza una tarea
Frames	10	...	Así se visualiza una tarea
	11		Así se visualiza una tarea

Eliminar un registro

La sentencia SQL **DELETE** se utiliza cuando necesitamos eliminar uno o más registros de una tabla relacional. Su manejo es muy simple:

```
DELETE FROM mistareas;
tr.executeSql('DELETE FROM mistareas');
```

Esta sentencia eliminará el total de registros que existan en la tabla en cuestión. Si queremos eliminar un registro puntual, podemos indicar su ID al momento de ejecutar la sentencia **SQL DELETE**:

```
tr.executeSql('DELETE FROM mistareas WHERE id = 8');
```

Esto eliminará el registro cuyo ID sea el 8 dentro de la tabla. Si creamos muchos registros con la sentencia anteriormente especificada, probemos la opción de eliminar un registro específico, indicando su ID mediante **WHERE**, tal como lo muestra la línea anterior.

Podemos ver en la imagen la ausencia del registro cuyo ID es el **número 8**, que hemos eliminado mediante **DELETE**.

	rowid	id	text
	1	1	Así se visualiza una tarea
	2	2	Así se visualiza una tarea
	3	3	Así se visualiza una tarea
	4	4	Así se visualiza una tarea
	5	5	Así se visualiza una tarea
	6	6	Así se visualiza una tarea
	7	7	Así se visualiza una tarea
	9	9	Así se visualiza una tarea
	10	10	Así se visualiza una tarea
	11	11	Así se visualiza una tarea

Leer y listar registros

Ya con una tabla alimentada de registros, solo nos queda ver de qué manera podemos leerlos con SQL para, luego, poder mostrarlos en el documento HTML en cuestión. Su sentencia es muy simple:

```
SELECT * FROM mistareas ORDER BY id;
```

Utilizando la sentencia **SELECT**, Web SQL nos devolverá todos los registros (esto está indicado mediante el *****, asterisco) de la tabla **mistareas**, ordenados mediante el campo **id**. De esta forma, conseguiremos listarlos tal como nos muestra la pantalla de herramientas para el desarrollador de Google Chrome.

	rowid	id	text
	1	1	Así se visualiza una tarea
	2	2	Así se visualiza una tarea
	3	3	Así se visualiza una tarea
	4	4	Así se visualiza una tarea
	5	5	Así se visualiza una tarea
	6	6	Así se visualiza una tarea
	7	7	Así se visualiza una tarea
	9	9	Así se visualiza una tarea
	10	10	Así se visualiza una tarea
	11	11	Así se visualiza una tarea
	12	12	Así se visualiza una tarea
	13	13	Así se visualiza una tarea
	14	14	Así se visualiza una tarea
	15	15	Así se visualiza una tarea
	16	16	Así se visualiza una tarea
	17	17	Así se visualiza una tarea
	18	18	Así se visualiza una tarea

Ahora, debemos aplicar esta sentencia **SELECT** dentro del código JavaScript, y también listar el resultado en pantalla. Lo primero que hacemos es ejecutar la consulta mediante **executeSql()**:

```
tr.executeSql('SELECT * FROM mistareas ORDER BY id', [], function (tr, results) {
    ...
});
```

En la ejecución de dicha consulta, si existen registros dentro de la tabla en cuestión, nos devolverá un array. Creamos una función que leerá y almacenará los resultados que necesitamos manipular.

Ahora, dentro de dicha función, declaramos dos variables: **len** y **html**, tal como lo muestra el siguiente código:

En la variable **len**, estamos guardando el total de los registros devueltos en **results**.

```
var len = results.rows.length, i;
var html = "";
```

La variable **html** la dejamos vacía, ya que la utilizaremos para armar el código HTML que tenemos que mostrar en el documento homónimo.

A continuación, declaramos una sentencia **For**, que recorrerá todos los registros existentes y armará el **html** para, finalmente, mostrarlos en pantalla:

```
for (i = 0; i <= len; i++) {
    ...
}
```

La sentencia **For** va desde **0** (cero) hasta el total de registros devueltos en la variable **len**. Por cada iteración que hace el **For**, alimentamos la variable **html**:

```
...
html = html + "<p>" + results.rows.item(i).id + " - " + results.rows.item(i).text
+ "</p>";
var posicion = document.getElementById("tareas");
posicion.innerHTML = html;
...
```

La variable **html** almacena una concatenación de textos y registros. Iniciamos un componente HTML del tipo **<p>** (paragraph); luego retornamos el objeto **results**, del cual leemos las filas que contiene (**rows**), y seleccionamos de ellas el **ítem(i).id**. Volvemos a concatenar con un espacio, guión y espacio, y repetimos la misma lectura del mismo registro, pero esta vez del ítem **text**. Finalmente, creamos la variable **posicion**, le asignamos el componente HTML cuyo **id** es **tareas**, y escribimos la variable **html** armada, en pantalla, utilizando **posicion.innerHTML = html**.

	rowId	id	text
1	1	1	Así se visualiza una tarea
2	2	2	Así se visualiza una tarea
3	3	3	Así se visualiza una tarea
4	4	4	Así se visualiza una tarea
5	5	5	Así se visualiza una tarea
6	6	6	Así se visualiza una tarea
7	7	7	Así se visualiza una tarea
8	8	8	Así se visualiza una tarea
9	9	9	Así se visualiza una tarea
10	10	10	Así se visualiza una tarea
11	11	11	Así se visualiza una tarea
12	12	12	Así se visualiza una tarea
13	13	13	Así se visualiza una tarea
14	14	14	Así se visualiza una tarea
15	15	15	Así se visualiza una tarea
16	16	16	Así se visualiza una tarea
17	17	17	Así se visualiza una tarea
18	18	18	Así se visualiza una tarea
19	19	19	Así se visualiza una tarea
20	20	20	Así se visualiza una tarea
21	21	21	Así se visualiza una tarea
22	22	22	Así se visualiza una tarea
23	23	23	Así se visualiza una tarea
24	24	24	Así se visualiza una tarea
25	25	25	Así se visualiza una tarea
26	26	26	Así se visualiza una tarea
27	27	27	Así se visualiza una tarea
28	28	28	Así se visualiza una tarea
29	29	29	Así se visualiza una tarea
30	30	30	Así se visualiza una tarea
31	31	31	Así se visualiza una tarea
32	32	32	Así se visualiza una tarea
33	33	33	Así se visualiza una tarea
34	34	34	Así se visualiza una tarea
35	35	35	Así se visualiza una tarea
36	36	36	Así se visualiza una tarea
37	37	37	Así se visualiza una tarea
38	38	38	Así se visualiza una tarea
39	39	39	Así se visualiza una tarea

Encriptar contenido

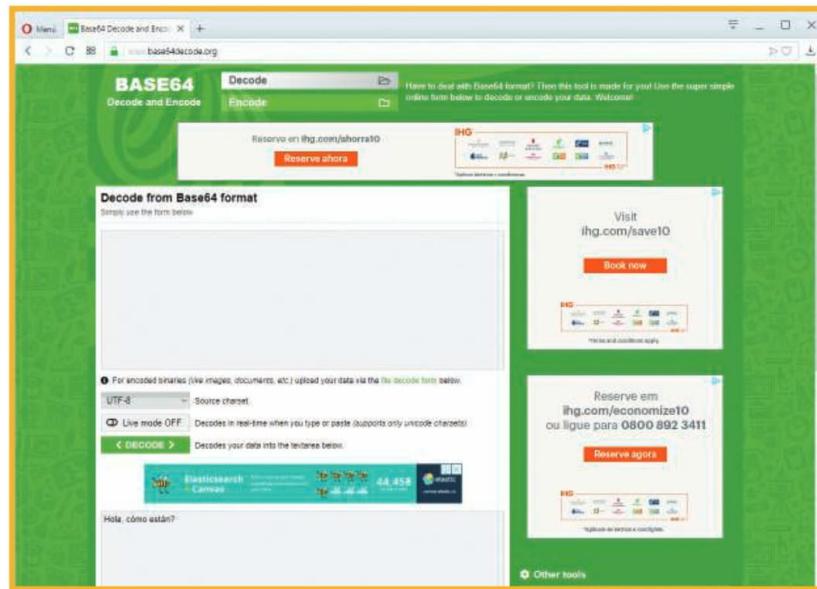


Una buena práctica que podemos implementar junto con el guardado de datos del lado del usuario es la encriptación de estos. JavaScript propone el estándar Base64 para hacerlo, que es lo que aprenderemos a utilizar a continuación y, si no nos gusta, veremos también cómo crear nuestro mecanismo de resguardo de la información.

Según [Wikipedia](#), Base64 es un sistema de numeración posicional que utiliza el número 64 como base. Este valor corresponde a la mayor potencia de dos que puede ser representada utilizando los caracteres **ASCII**.

Más información en: <https://es.wikipedia.org/wiki/Base64>.

El lugar donde comúnmente se aplica este mecanismo de codificación es en los sistemas de correo electrónico. Por lo general, el contenido adjunto a un mail (archivos, imágenes, etcétera) se comprime al formato Base64 para transformarlo en caracteres y así poder enviarlo de manera incrustada dentro del mensaje.



Existe en el mundo una organización que se ocupa de la difusión y el mantenimiento de este estándar, cuyo sitio web es www.base64decode.org. Allí encontraremos toda la información necesaria para entender a fondo las directrices de este estándar, y también podremos hacer pruebas de codificación y decodificación de información.

Seguridad con Base64

JavaScript incluye en la sintaxis de su lenguaje métodos ya armados que nos permiten codificar y decodificar contenido en formato Base64. Los dos métodos asociados para realizar estas tareas son:

- **atob()**: encripta caracteres ASCII a Base64
- **btoa()**: desencripta Base64 a caracteres ASCII

Función que recibe un parámetro del tipo texto, y codifica a Base64

Su uso es muy simple: solo debemos llamar al método, pasándole el texto que deseamos codificar o viceversa, para luego almacenar el resultado en una variable y guardarla en una base de datos remota o un sistema de almacenamiento de datos local.

btoa(texto);

Parámetros que reciben los métodos de encriptación y desencriptación.

atob(texto_encriptado);

Función que recibe el parámetro encriptado en Base64, y lo decodifica a texto ASCII.

Encriptar con btoa()

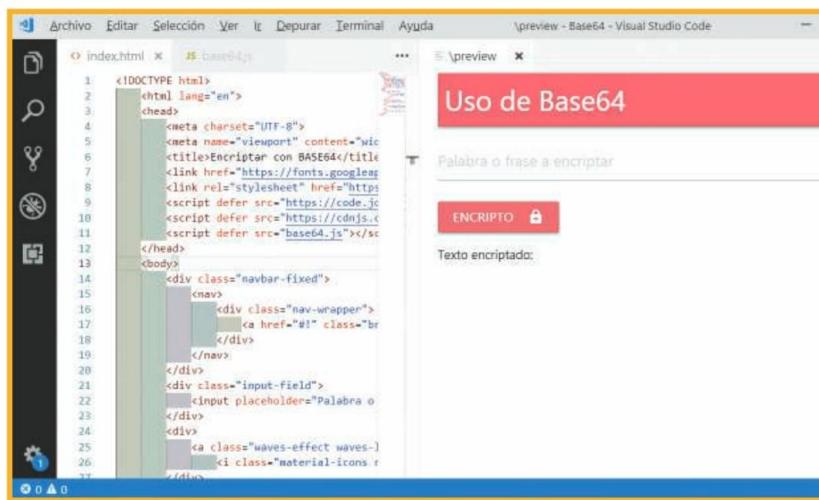
Realizaremos a continuación el ejercicio de encriptar contenido. Para esto, descargamos el archivo **Ejercicio-Base64.zip** del repositorio de material adicional que contiene esta obra. Luego, creamos un nuevo proyecto en Visual Studio Code, y agregamos el documento HTML que contiene el material descargado.



Si ejecutamos una vista previa del archivo HTML, veremos que este simplemente contiene un campo de texto, un botón y una etiqueta del tipo Paragraph.

El campo cuyo ID es **texto** permitirá que escribamos un texto o frase para encriptar.

El botón denominado **botón-encriptar** ejecutará la función JS que realizará el trabajo, y dentro del Párrafo **texto-resultado**, visualizaremos el contenido encriptado en formato de caracteres.



Texto encriptado:

```
bar-fixed">

    ass="nav-wrapper">
    href="#" class="brand-logo" :>
```

out-field">
 placeholder="Palabra o frase a encriptar" :>

waves-effect waves-light btn-:
 s="material-icons right">encriptar

Crear archivo

×

No se puede abrir 'base64.js': Archivo no encontrado
(file:///c:/Users/fluna/Desktop/Proyecto
11/Fuentes/Base64/base64.js).

Ubiquemos en el código del archivo HTML la declaración del archivo JS base64.js, y hagamos CTRL + clic sobre él, aceptando a continuación su creación.

Ahora declaramos en el archivo JS las primeras sentencias de nuestra aplicación web de encriptación:

```
document.addEventListener("DOMContentLoaded", function() {
    document.getElementById("boton-encriptar").addEventListener("click", function() {
        ...
    });
});
```

Con esto nos aseguramos de que el script solo se ejecutará cuando el HTML se haya cargado completamente, y la función de encriptación la ejecutaremos cuando el usuario pulse **botón-encriptar**.

Luego, agregamos el siguiente código dentro de la función del botón:

En la variable **palabra** almacenamos el contenido escrito en el campo de texto.

Mediante una sentencia **If**, nos aseguramos de que dicho campo tenga un contenido escrito antes de ejecutar la función de encriptación.

Si no tiene contenido, alertamos al usuario que debe escribir algo; y si lo tiene, ejecutamos la función **encripto()** pasándole **palabra** como un parámetro.

```

var palabra = document.getElementById("texto").value;
if (palabra === "") {
    alert("Debe ingresar un texto válido.");
    palabra.focus;
} else {
    encripto(palabra);
}

```

Declarar la función de encriptación

A continuación, declararemos la función de encriptación y llamaremos al método **btoa()** para que haga lo propio con el texto pasado como parámetro:

```

function encripto(texto) {
    var texto_codificado = btoa(texto);
    document.getElementById("textoResultado").innerText = texto_codificado;
}

```

```

index.html          JS base64.js
1  document.addEventListener("DOMContentLoaded", function() {
2
3      document.getElementById("boton-encriptar").addEventListener("click", function() {
4          var palabra = document.getElementById("texto").value;
5          if (palabra === "") {
6              alert("Debe ingresar un texto válido.");
7              palabra.focus;
8          } else {
9              encripto(palabra);
10         }
11     });
12
13     function encripto(texto) {
14         var texto_codificado = btoa(texto);
15         document.getElementById("textoResultado").innerText = texto_codificado;
16     }
17 });

```

En esta función, declaramos la variable **texto_codificado**, donde almacenamos el resultado de la función **btoa()**. Finalmente escribimos el **label texto-resultado** con el resultado codificado en base64. Ya tenemos las condiciones adaptadas para probar nuestro sistema de encriptación:

Probemos diferentes palabras o frases completas, para ver el comportamiento correcto de nuestro primer sistema de encriptación de información.

Ahora, ¿se animan a guardar este resultado localmente, utilizando **LocalStorage**?



Desencriptar datos con `atob()`

El procedimiento ahora es a la inversa, para ver lo que hemos codificado anteriormente, en formato estándar ASCII. Para esto, vamos a modificar el documento HTML, agregando un botón denominado **boton-desencriptar**, y un nuevo label llamado **texto-desencriptado**. Luego añadimos:

```
document.getElementById("boton-desencriptar").addEventListener("click",
function() {
    var texto_encriptado = document.getElementById("texto-resultado").innerText;
    if (texto_encriptado === "") {
        //Nada para desencriptar
    } else {
        desencripto(texto_encriptado);
    }
})
```

El diseño debe quedar similar al de la imagen.



Por último, creamos la función correspondiente que desencripta el dato:

```
function desencripto(base) {
    var texto = atob(base);
    document.getElementById("texto-desencriptado").innerText = texto;
}
```

Ya con el resultado final, podemos ejecutar este ejercicio en el navegador web y comenzar a probarlo.



Para soluciones de encriptación mucho más efectivas, podemos recurrir a librerías o frameworks como **WebCryptoAPI**, **sjcl**, **js-nacl**, **jsencrypt**, **OpenPGP.js**, **jwcrypto**, o similares. Estas nos darán mucha mayor efectividad al momento de proteger información sensible.

Tengamos siempre presente que Base64 no es un sistema seguro de encriptación, y solo nos servirá para dar un nivel mínimo de protección a los datos registrados del lado del usuario o a aquellos datos que deban viajar por un formulario web hacia nuestro backend web.

Codificar nuestro sistema de encriptación

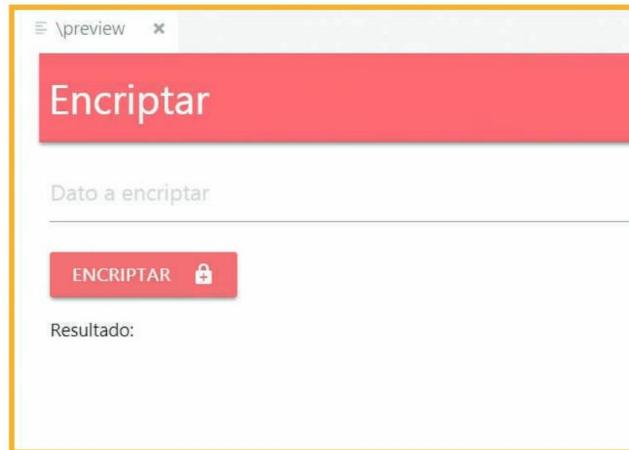
Si no queremos utilizar Base64 como mecanismo de encriptación, ni tampoco emplear las librerías y frameworks JS que existen actualmente, podemos programar nuestro propio sistema de encriptación, que ayudará a almacenar datos más seguros, tanto del lado del usuario como en forma remota. Veamos cómo hacerlo.

Esta solución requiere de bastante código fuente, por lo que será más efectivo descargar el ejemplo directamente del repositorio de archivos de esta obra:

Ejemplo-Encriptacion-Completo.

zip. Creemos a continuación un nuevo proyecto y agreguemos estos archivos a él. Finalmente, ejecutemos el documento HTML para ver su estructura gráfica.

Como podemos visualizar en la imagen, el código HTML es similar al que trabajamos en el ejercicio de Base64, y lo único que cambia en este caso es el contenido del archivo **encriptacion.js**. ¿Cómo resolvemos la encriptación? Vamos más allá de la codificación de caracteres del tipo ASCII, y utilizamos en su lugar caracteres extendidos, de esos que podemos encontrar dentro de la



aplicación **Mapa de Caracteres** que viene en Windows, o en el sistema operativo que utilicemos. Si editamos dicho código, lo que hacemos, básicamente, en el mecanismo de encriptación es asociar con cada carácter ASCII, un carácter extendido de uno o dos bytes, dependiendo de nuestra elección.

Veamos a continuación un ejemplo:

```

case "a": iterar = "Ijx"; break; //IJ
case "b": iterar = "Hg"; break; //H
case "c": iterar = "bç"; break; //ç
case "d": iterar = "Bt"; break; //Б
case "e": iterar = "əH"; break; //Ә
case "f": iterar = "Ły"; break; //Ł
case "g": iterar = "Qh"; break; //Ң
case "h": iterar = "ħñ"; break; //ħ
...

```

```

case "V": iterar = "yЧ"; break; //Ч
case "W": iterar = "EĽ"; break; //Ľ
case "X": iterar = "ęk"; break; //ę
case "Y": iterar = "|n"; break; //ń
case "Z": iterar = "łš"; break; //ł
case "1": iterar = "0Ń"; break; //ń
case "2": iterar = "Sđ"; break; //đ
case "3": iterar = "œf"; break; //ƒ
case «4»: iterar = «JħK»; break; //ħ

```

De esta manera, reemplazamos por un carácter extendido cada carácter del alfabeto en minúscula, mayúscula, números, vocales con tilde y símbolos operacionales del pad numérico del teclado. Si se ingresan caracteres extendidos, estos serán respetados tal cual fueron ingresados, anteponiendo un símbolo delante y colocando otro detrás, para que nuestro script lo identifique y elimine los símbolos en cuestión al momento de decodificarlo.

El mecanismo de encriptado

Para encriptar un texto o frase ingresados en el campo de texto, declaramos una función que recibe como parámetro el contenido a encriptar. El siguiente paso será leer carácter por carácter el parámetro a encriptar y así buscar su equivalente en la tabla de caracteres que creamos previamente.

Esto último lo realizamos mediante un bucle **For**, el cual recorre desde el primer carácter hasta el último; y la función **charAt()**, alojada en cada variable de texto JS, que permite obtener la posición del carácter que necesitamos.

```

case "í": iterar = "L@"; break; //
case "ó": iterar = "ea"; break; //
case "ú": iterar = "-w"; break; //
default:
    var mascaracaracteres = " ,;:-{[~`+~*]}`{|?\"=)(/&%$#!°|¬"; var r;
    var r = Math.random() * (0, mascaracaracteres.length);
    iterar = "//" + azar + mascaracaracteres.charAt(r);
}
pwdEncriptado = pwdEncriptado + iterar;
}
console.log("La codificación realizada dio como resultado: " + pwdEncriptado);
return pwdEncriptado;
}

```

Una vez obtenido el carácter, ejecutamos la sentencia **SWITCH – CASE**, para buscar su equivalencia dentro de los caracteres extendidos que creamos, y reemplazar el carácter inicial por su equivalente de encriptación. Al final del bucle, se va armando en la variable **pwdEncriptado** todo el reemplazo de caracteres que realizamos a través de la función de codificación. Finalmente, podemos probar el resultado de encriptación ingresando en la caja de texto del documento HTML cualquier contenido que deseemos llevar a una codificación. En el componente **label** contiguo, veremos el resultado con nuestro propio mecanismo.



Detectar conectividad a Internet



Dentro de las tantas bondades del lenguaje JavaScript, podemos encontrar la capacidad de detectar si hay o no conectividad a Internet. Esta práctica es muy útil para implementar en la mayoría de las aplicaciones de misión crítica y en las aplicaciones web móviles, antes de una operación de envío u obtención de datos remotos.

Una de las grandes opciones que ofrece JavaScript para detectar determinados parámetros del ambiente virtual donde se está ejecutando una aplicación web es el objeto **Navigator**. Este incluye numerosas posibilidades (algunas muy buenas, como saber si estamos online o no), que nos permiten, entre otras cosas, conocer: sistema

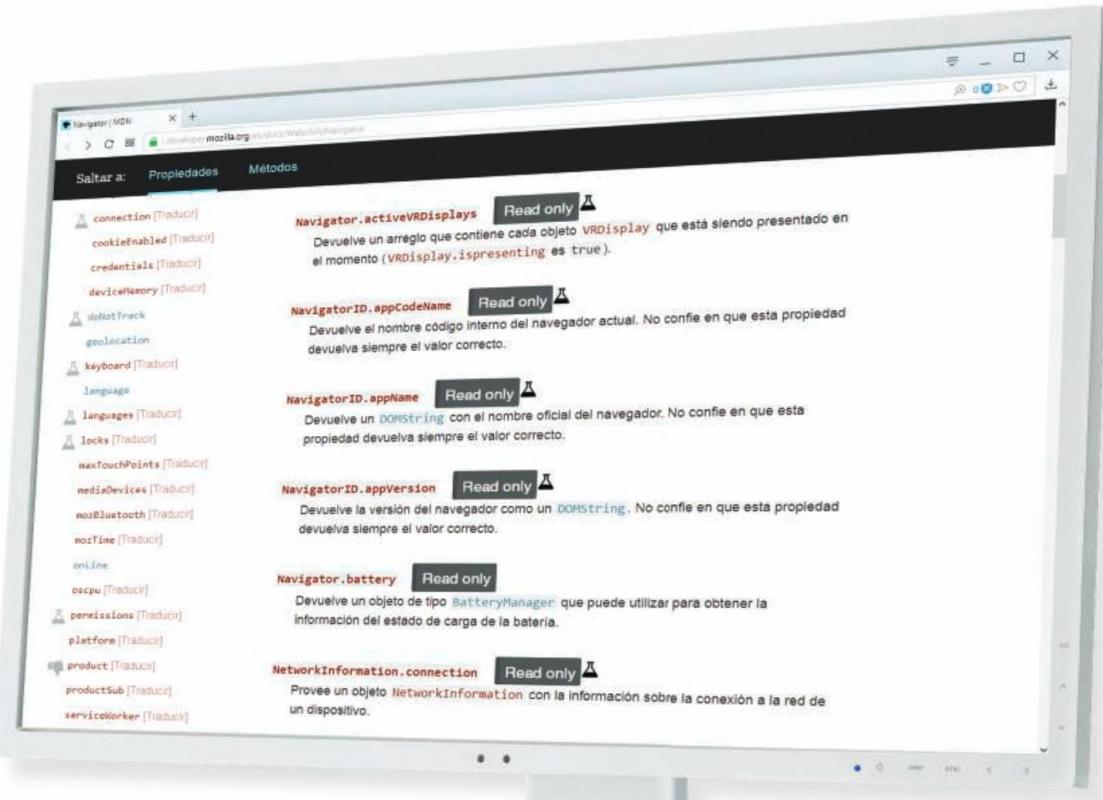
operativo, tipo de navegador y plataforma de hardware.

Muchas de estas opciones se incluyen en el lenguaje casi desde sus inicios, mientras que otras son muy nuevas o hasta experimentales. Veamos a continuación una lista de ellas, para tenerlas presentes en caso de necesitar saber algún dato crucial que ellas nos puedan aportar.

El objeto Navigator

PROPIEDAD	DESCRIPCIÓN
cookieEnabled	Retorna TRUE si las cookies están activas, o FALSE en caso contrario.
userAgent	Devuelve el UserAgent del navegador web.
language	Devuelve el idioma de la aplicación (solo Firefox y Chrome).
plugins	Devuelve el listado de plugins instalados (solo en Firefox y Chrome).
systemLanguage	Devuelve el idioma del sistema operativo (solo en IE y Edge).
Geolocation	Retorna la ubicación GPS del usuario del navegador web.
platform	Retorna la plataforma del sistema operativo.
online	Devuelve TRUE si hay conectividad a Internet o, de lo contrario, FALSE.

Existen numerosas opciones incluidas en el objeto **Navigator**, pero varias de ellas son propietarias de un navegador web determinado, y otras tantas han sido desestimadas y solo están activas por retrocompatibilidad.



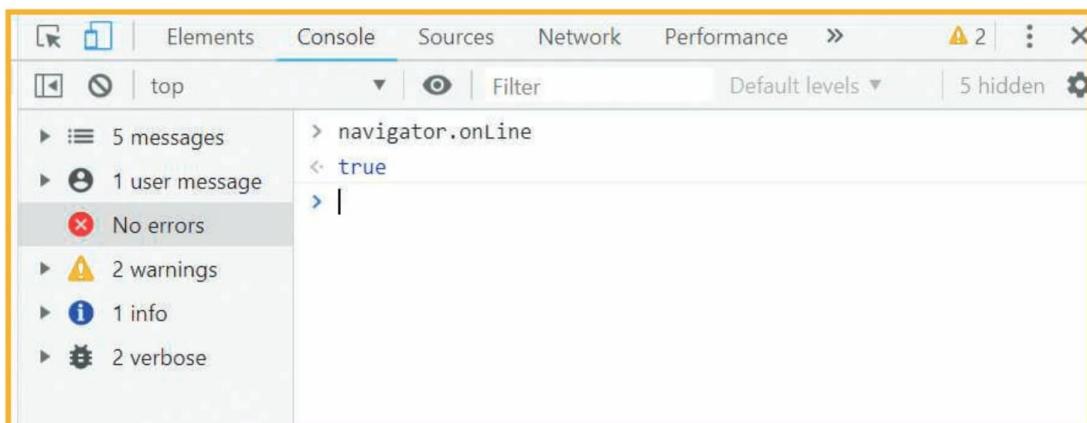
En Mozilla Developer Network podemos investigar sobre todas las opciones que nos ofrece el objeto Navigator, con información actualizada: <https://developer.mozilla.org/es/docs/Web/API/Navigator>.

navigator.onLine

Como vemos en la tabla anterior, el objeto **navigator**, seguido de la propiedad **onLine**, devuelve un valor booleano que indica si el web browser está o no conectado a Internet. Para probarlo, abramos nuestro navegador web y accedamos a las herramientas para el desarrollador. Luego, escribamos en la consola:

```
navigator.onLine
```

Al presionar ENTER, el navegador debe devolver un valor **TRUE** si estamos conectados a Internet, o **FALSE** en el caso contrario.



/<Ejercicio práctico>

Detección de conectividad

Realicemos a continuación un ejercicio que nos permita llevar a la práctica esta funcionalidad tan útil que propone JavaScript. Comencemos por descargar el archivo **Proyecto-Conectividad-Base.zip** del repositorio de material de esta obra. En él encontraremos un documento HTML con una interfaz simple, tal como se observa en la imagen. Esta interfaz cuenta con un ícono de conectividad, un botón y una barra de progreso, en principio, oculta. La funcionalidad que estableceremos mediante JavaScript es detectar si estamos online u offline. En el caso de estar online, la barra de progreso estará oculta, el botón (+) estará habilitado con su correspondiente funcionalidad, y el ícono de conectividad indicará que está funcional. En el momento en que perdemos la conectividad, el ícono cambiará a modo Desconectado, la barra de progreso será visible a la espera de volver a



conectarse y el botón (+) deshabilitará su función **click**. Hagamos **CTRL + clic** dentro del documento HTML sobre la referencia JS **chequearConexion.js**. Aceptemos la creación del archivo y escribamos el siguiente código:

```
document.addEventListener("DOMContentLoaded", function () {
    validarConectividad();
})
documento.getElementById("boton-agregar").addEventListener("click", function() {
    alert("Hola mundo!");
})
```

Mediante **AddEventListener** detectamos que el documento HTML se haya cargado por completo, y allí invocamos la función **validarConectividad()**, la cual aún debemos crear. Por otro lado, le damos una simple interactividad al botón a través de la función **alert()**, que solo podremos ejecutar cuando el navegador esté online. Vamos ahora a crear la función **validarConectividad()**, escribiendo lo siguiente a continuación del código anterior:

```
function validarConectividad() {
    setInterval(function() {
        var conectado = navigator.onLine;
        if (conectado) {
            habilitoComponentes();
        } else {
            deshabilitoComponentes();
        }
    }, 4000
);}
```

Al entrar en esta función, ejecutamos JS **setInterval()** cada 4 segundos, chequeando conectividad. Luego, en la variable **conectado** guardamos el estado devuelto por **navigator.onLine**. Si estamos conectados, llamamos a la función **habilitoComponentes()** y, de lo contrario, llamamos a la función **deshabilitoComponentes()**. Estas últimas dos funciones debemos crearlas, seguidas a este último bloque de código:

```

function habilitoComponentes() {
    document.getElementById("senial").setAttribute("class", "material-icons visible");
    document.getElementById("no-senial").setAttribute("class", "material-icons hide");
    document.getElementById("barra").setAttribute("class", "progress hide");
    document.getElementById("boton-agregar").setAttribute("class", "btn-floating
btn-large waves-effect waves-light red right shadow");
    console.info("Houston, nos hemos conectado con éxito!");
}

```

La función simplemente llama al cambio de determinados atributos, denominados Clases, de cada componente HTML. Ocultamos el ícono Desconectado, ocultamos la barra de progreso y habilitamos el botón (+). Ahora haremos lo propio con la otra función:

```

function deshabilitoComponentes() {
    document.getElementById("senial").setAttribute("class", "material-icons hide");
    document.getElementById("no-senial").setAttribute("class", "material-icons visible");

    document.getElementById("barra").setAttribute("class", "progress visible");
    document.getElementById("boton-agregar").setAttribute("class", "btn-floating
btn-large waves-effect waves-light grey disabled right shadow");
    console.error("Houston, tenemos un problema... de conectividad!");
}

```

Cómo probar conectividad desde un dispositivo móvil

Ya tenemos listo el código de nuestra aplicación web. Solo resta probarlo, y para hacerlo, podemos subirlo a un servidor web, o ejecutar dentro de nuestra red WiFi un web server que contenga estos archivos y que pueda ser accedido desde un dispositivo móvil.

En el caso de esta última opción, luego de cargar la página web en el navegador móvil, ponemos el equipo en **Modo Avión** y esperamos que ocurra el cambio de componentes HTML en pantalla dentro del margen de los 4 segundos. Luego, repetimos el mismo procedimiento, activando otra vez la conectividad.

Cómo probar conectividad desde una computadora

Desde una computadora, tenemos dos opciones para las pruebas. Una de ellas es poner el equipo en Modo Avión (una característica que incluyen los sistemas operativos Windows 8 o posterior, Linux y OS-X), y la otra es usar las herramientas para el desarrollador de Google Chrome. En este último caso, cargamos nuestro proyecto web en Chrome, presionamos **F12**, y dentro de las herramientas para el desarrollador, ubicamos la pestaña **Network**

y luego la subpestana **Network Conditions**. De manera predeterminada, veremos que nuestro proyecto HTML se ejecuta sin problemas, detectando la conectividad necesaria. Ahora, desplegamos el combo de opciones **Network Throttling** y seleccionamos la opción **Offline**. Así, nuestro código detectará la falta de conectividad, y actualizará los componentes HTML en pantalla, según lo indicado en el algoritmo que hemos escrito..

Adaptar gráficos y multimedia según la performance



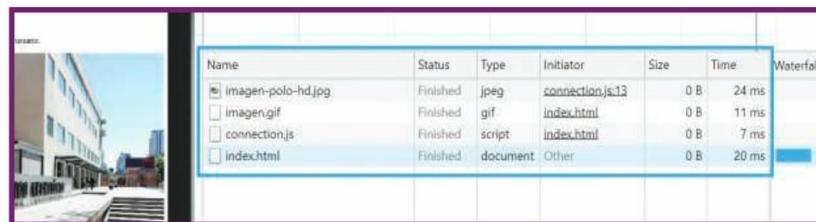
Este nuevo proyecto nos permitirá cargar imágenes y videos optimizados, según la velocidad de navegación o el poder de procesamiento del hardware de quienes visitan nuestra web.

Antes que comenzar, debemos aclarar que las propiedades del objeto **Navigator** que analizaremos a continuación se encuentran actualmente en modo experimental. ¿Qué significa esto? Quiere decir que están presentes de modo parcial en algunos navegadores web, y que los respectivos desarrolladores de estas características están tratando que la **W3C** (el consorcio que administra los estándares web) las

ponga oficialmente en vigencia. Creemos conveniente aprovechar estas características, que seguramente terminen por hacerse efectivas con el tiempo. ¿Y qué pasa si esto último no ocurre? Por lo general, aquellos navegadores que las incluyeron en su motor web las dejan vigentes durante muchos años, o hasta que salga un estándar web oficial que se aplique de forma masiva.

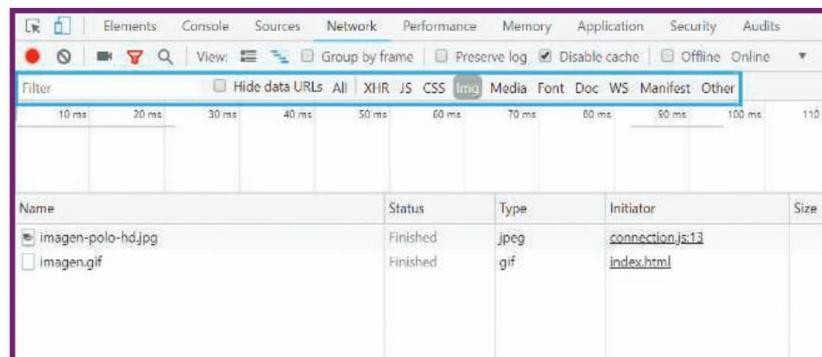
Medir la performance de carga

1 A través de las Herramientas para el desarrollador, Google Chrome nos permite medir la velocidad de carga de un sitio web, analizando todos sus componentes, y visualizándolos juntos o solo algunos de ellos.



2 Para esto, debemos ingresar en el apartado **Network** de estas herramientas y cargar la página o sitio que deseamos. En el extremo superior de la grilla de análisis, encontraremos la posibilidad de filtrar los componentes a analizar, pudiendo elegir entre todos, imágenes, multimedia, JavaScript, CSS, etcétera.

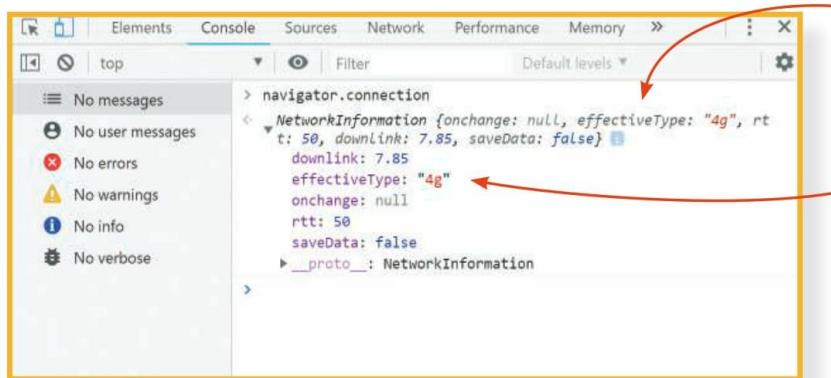
3 Si bien no hay un estándar definido para la carga óptima y veloz de un sitio web (dado que existe en el mundo un sinfín de velocidades de navegación que varían según el tipo de conexión y dispositivo con el cual accedemos a Internet), lo ideal es siempre tener cada web lo más optimizada posible para que el usuario no deba esperar mucho tiempo por la respuesta del sitio que quiere ver. Teniendo esto presente, comenzemos a trabajar el contenido para la primera fase de este proyecto.



navigator.connection

Como bien mencionamos anteriormente, el objeto Navigator cuenta con más propiedades que permiten llevar a cabo otras tareas específicas del navegador, entre ellas, evaluar la conexión del usuario. Podemos comenzar a probar esto mismo en la consola de Chrome. Presionamos la tecla **F12** y a continuación la pestaña **Console**. Allí, en su línea de comandos, escribimos:

```
Navigator.connection;
```



El resultado que nos devuelve es un objeto **JSOn** (más adelante analizaremos qué es) con una serie de datos, tal como lo muestra la imagen.

Dentro del conjunto de valores que retorna, podemos ubicar uno denominado **effectiveType**; este es el que almacena el tipo de conexión que el usuario utiliza. También podemos escribir la sentencia anterior seguida de un punto + **effectiveType**, para llegar directamente al valor que necesitamos analizar.

Valores que devuelve

Los valores de conectividad que suele devolver **navigator.connection** son:

- 4g
- 3g
- 2g

Al momento de escribir esta publicación no existen otras opciones que esta funcionalidad pueda devolver. Entendemos que **4g** engloba todas las conexiones móviles 4g junto con las conexiones **WiFi**, **cablemódem**, **ADSL** y **Ethernet**. Bajo **3g**, se incluyen las conexiones móviles en general, y **2g** hace referencia a conexiones de baja velocidad, como **GPRS**, cablemódem y ADSL de 1 Mbps o menos.

/<Ejercicio práctico>

Cargar una imagen optimizada

A continuación, descargamos el archivo **Ejercicio-navigator.connection.zip**.

Descomprimimos los archivos contenidos en una carpeta y creamos un proyecto en VS Code, seleccionando dicha carpeta. Dentro de este contenido, encontraremos una subcarpeta **images** con tres archivos **JPG** y un archivo **GIF**, tal como muestra la imagen.



Las tres imágenes JPG son el mismo archivo, recortado y optimizado en diferentes tamaños (en Kilobytes). Lo que haremos en este ejercicio es analizar con qué velocidad se conecta el usuario y, según este dato, le enviaremos uno u otro archivo de mayor o menos peso.

En el documento HTML tenemos contenido estándar: un título **<h1>**, un párrafo **<p>** y una imagen ****. Esta última tiene asociado el archivo **GIF** de forma predeterminada, por si por algún motivo no se carga ninguna de las otras imágenes.

```

1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta charset="utf-8" />
5     <meta http-equiv="X-UA-Compatible" content="IE=edge">
6     <title>Imágenes optimizadas</title>
7     <meta name="viewport" content="width=device-width, initial-scale=1, user-scalable=no">
8     <script src="connection.js" defer></script>
9   </head>
10  <body>
11    <h1>Imágenes optimizadas</h1>
12    <p>Cargamos una imagen optimizada, de acuerdo al tipo de conexión del usuario.</p>
13    <p></p>
14    
15  </body>
16 </html>

```

Creamos el archivo JavaScript con CTRL + clic sobre la declaración en el HTML. Declaramos en primera instancia el AddEventListener que espera que cargue el contenido completo:

Luego creamos dos variables: **foto**, que asociamos con el componente HTML de la imagen; y **tconn**, que guardará el tipo de conexión del usuario:

```

document.addEventListener("DOMContentLoaded", function() {
  ...
});

```

```

var foto = document.getElementById("imagen");
var tconn = navigator.connection.effectiveType;

```

Creamos a continuación nuestro archivo JavaScript, haciendo CTRL + clic Finalmente, utilizamos la sentencia **Switch** para analizar qué valor nos ha devuelto **effectiveType**. De acuerdo con este dato, cargamos una u otra imagen utilizando **foto.setAttribute**, donde asignaremos al atributo **src** el **nombre de la imagen** que deseamos visualizar:

```

switch (tconn) {
  case "2g":
    foto.setAttribute("src", "images/imagen-polo-pequenia.jpg");
    break;
  case "3g":
    foto.setAttribute("src", "images/imagen-polo-normal.jpg");
    break;
  case "4g":
    foto.setAttribute("src", "images/imagen-polo-hd.jpg");
    break;
  default:
    foto.setAttribute("src", "images/imagen-polo-pequenia.jpg");
    break;
}

```

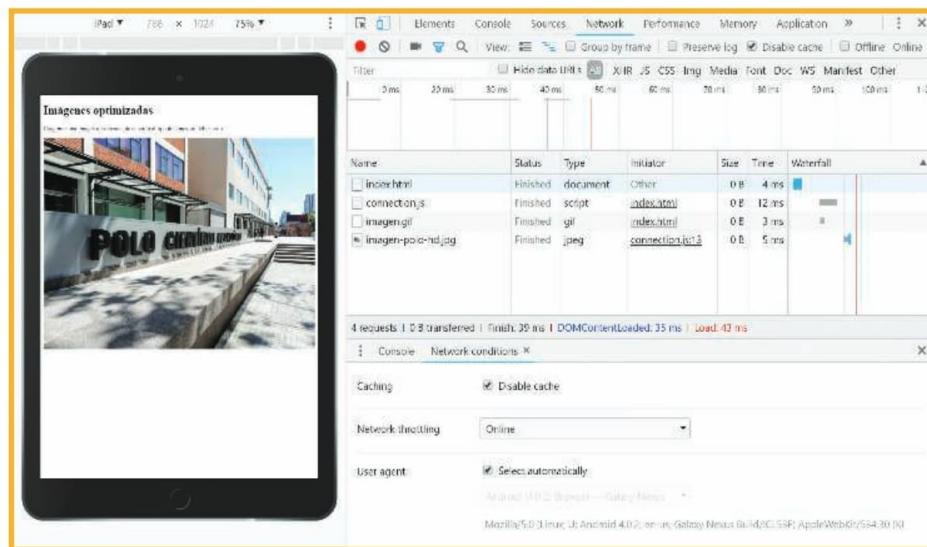
Ejecutemos a continuación el proyecto dentro del navegador web Chrome de escritorio.



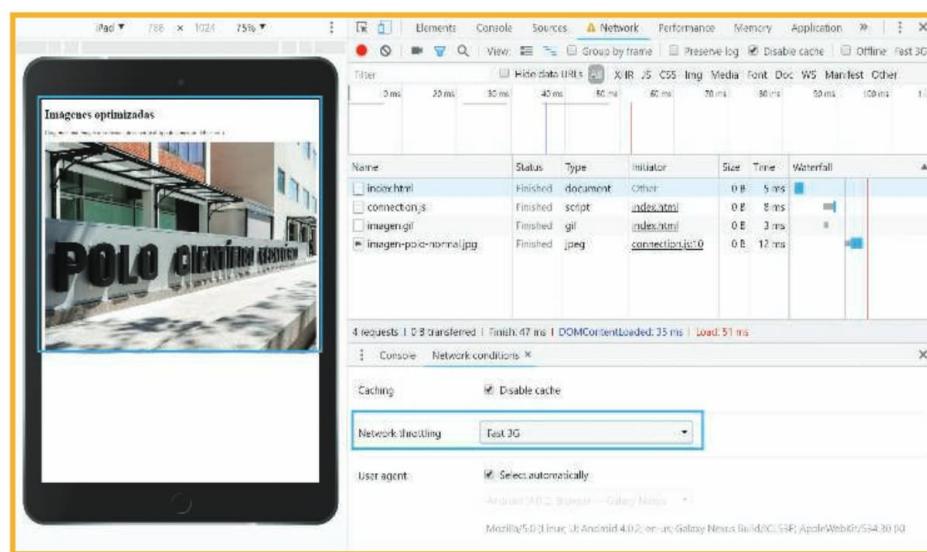
Si todo salió bien, y disponemos de una conexión que se englobe dentro de 4G, veremos que la página web cargó la imagen de mayor resolución junto con el documento HTML.

Simular diferentes velocidades de conexión

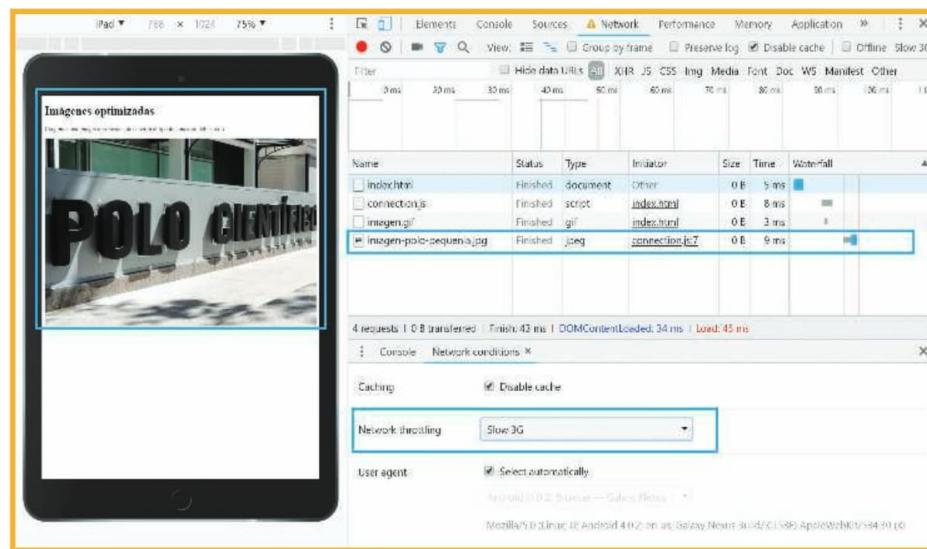
Veamos ahora cómo, a través de Chrome, podemos simular diferentes velocidades de conexión y, así, evaluar el correcto comportamiento de nuestra solución web. Dentro de Google Chrome, pulsamos la tecla F12 para iniciar las Herramientas para el desarrollador, nos aseguramos de cargar el documento HTML que trabajamos hasta el momento y nos posicionamos en la pestaña **Network**.



En la subpestana **Network Conditions**, ubicada en el extremo inferior de la grilla de carga, está el combo box **Network throttling**, que por defecto debe decir **Online**. Lo desplegamos, seleccionamos **Fast 3G** y recargamos la página presionando **CTRL + R**.



Por último, nos queda volver a cambiar el valor de red por **Slow 3g**, que es el equivalente a **GPRS** o el antiguo **2g**. Recargamos la página, y allí podemos ver que la imagen vuelve a cambiar. Si analizamos los tiempos de carga de la imagen principal, notaremos que van disminuyendo.



navigator.hardwareConcurrency

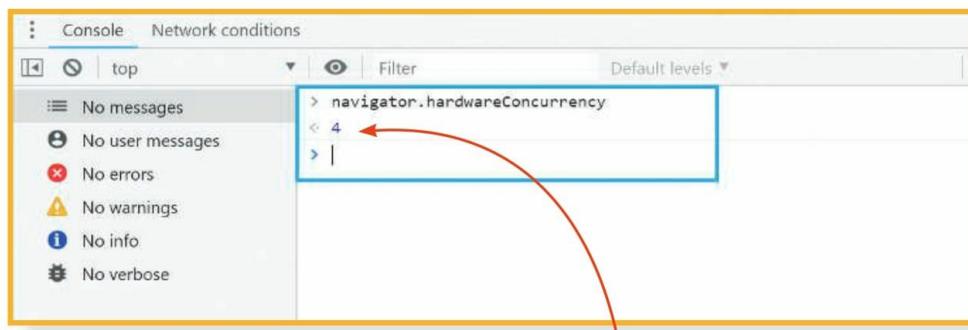
Veamos a continuación de qué manera podemos evaluar la clase de hardware con el que accede cada cliente y, de esa forma, determinar el tipo de multimedia más acorde a enviar a su dispositivo.

Cuando manejamos información multimedia, ya sea video o música a través de streaming, debemos analizar no solo el tipo de velocidad de navegación del dispositivo móvil o la computadora de cada cliente, sino también la capacidad de procesamiento que este tiene.

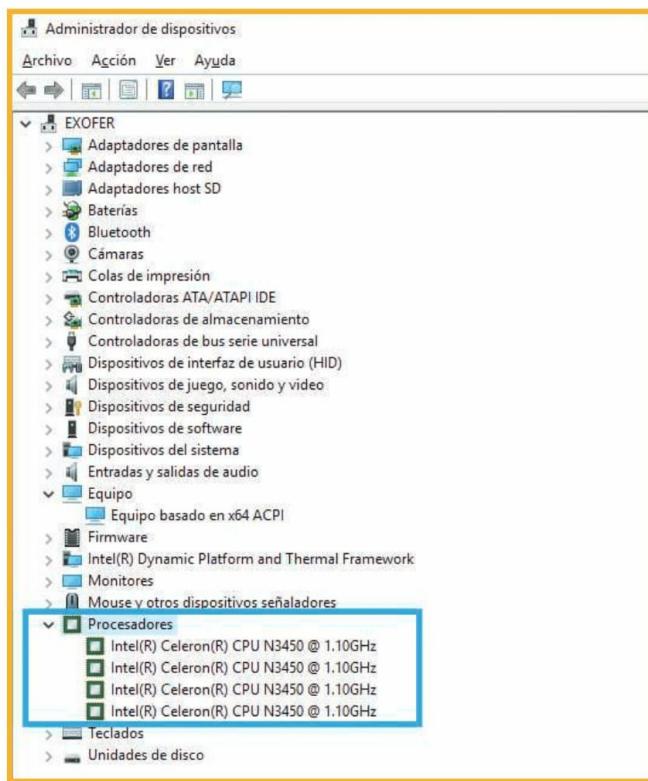
Por eso, a través de la propiedad **hardwareConcurrency**, también incluida dentro del objeto **Navigator**, podremos analizar, junto con **navigator.connection.effectiveType**, cuál es el contenido más exacto que el dispositivo del usuario podrá reproducir. Entonces, simplemente invitamos la sentencia:

```
navigator.hardwareConcurrency;
```

Como resultado, obtendremos un número entero, tal como lo muestra la imagen.



¿Qué significa ese valor? Equivale a la cantidad de hilos o subprocessos que puede desarrollar el microprocesador de un dispositivo móvil o computadora. Para comparar esto en nuestro equipo, solo debemos acceder a las propiedades del sistema y ubicar la información referente al procesador usado. Allí podremos visualizar claramente cuántos hilos puede manejar el procesador de nuestra máquina, y compararlo con el resultado que nos devuelve esta sentencia JavaScript.



La implementación de este tipo de características puede encontrarse en plataformas como **YouTube** o **Vimeo**, entre otras. Al momento de cargar sus páginas, estas analizan el tipo de procesador que tiene el usuario y entonces le ofrecen una experiencia limitada o ilimitada en cuanto a calidad de video. Por ejemplo, YouTube exige un procesador de al menos doble núcleo (2 hilos) para poder reproducir sus videos. Si ingresamos con un mononúcleo, la plataforma no nos permitirá reproducir ningún tipo de video.

¿Por qué no resolver la calidad de video con la información de navigator.connection?

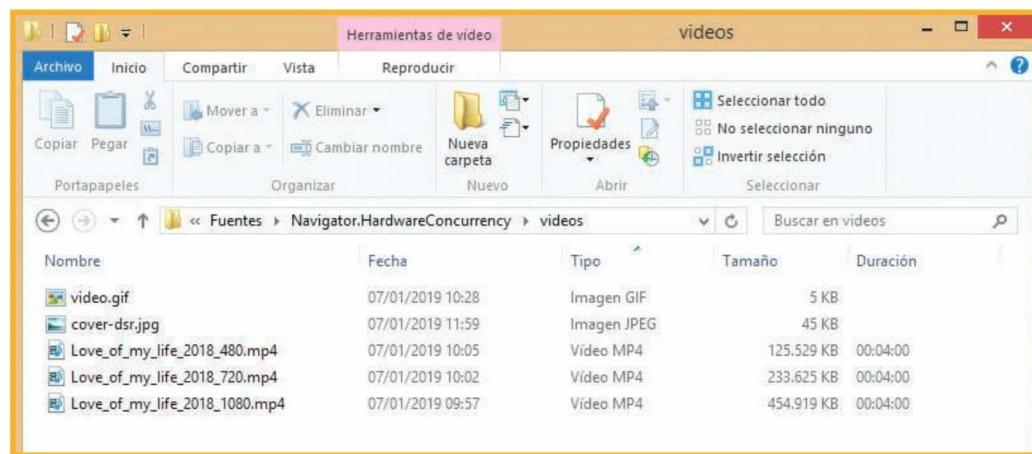
Si bien podemos optar por enviar un video con determinada calidad (dependiendo de si el usuario se conecta por 2g, 3g o 4g), esto no es suficiente, ya que si el hardware de nuestro visitante tiene solo uno o dos núcleos y, además, se está conectando mediante 3g, cargar un video

FullHD de al menos 10 minutos le demandará un tiempo más que considerable y se entrecortará de forma constante. En cambio, si el procesador del usuario es de dos núcleos, pero se conecta a 4g, podremos ofrecerle un video de calidad HD, y un tiempo de carga medio.

/<Ejercicio práctico>

Entregar la mejor calidad de video

Haremos ahora algunos agregados a nuestro proyecto anterior. Lo primero será incorporar al proyecto anterior una carpeta con videos a diferente resolución, más dos imágenes de portada. Para esto, descargamos el archivo **videos-hardwareConcurrency.zip**, lo descomprimimos y agregamos la carpeta video como subcarpeta del proyecto **Navigator.connection**. En esta carpeta veremos el contenido que se detalla en la **imagen**: videos a 1920x1080, 1366x720 y 854x480 píxeles, más dos imágenes: **cover-dsr.jpg** y **video.gif**.



A continuación, agregamos el siguiente código a nuestro documento HTML, justo debajo de la imagen visualizada:

```
...
<hr/>
<h1>Videos optimizados</h1>
<video id="video" src="" poster="images/video.gif" width="480"
controls></video>
<p id="alerta" style="background-color:red;color:white;"></p>
</body>
</html>
```

El contenido del documento HTML debe quedar estructurado como en la imagen.

```

index.html x JS connection.js
1  <!DOCTYPE html>
2  <html>
3      <head>
4          <meta charset="utf-8" />
5          <meta http-equiv="X-UA-Compatible" content="IE=edge" />
6          <title>Imágenes optimizadas</title>
7          <meta name="viewport" content="width=device-width, initial-scale=1, user-scalable=no" />
8          <script src="connection.js" defer></script>
9      </head>
10     <body>
11         <h1>Imágenes optimizadas</h1>
12         <p>Cargamos una imagen optimizada, de acuerdo al tipo de conexión del usuario</p>
13         <p></p>
14         <hr/>
15         <h1>Videos optimizados</h1>
16         <video id="video" src="" poster="images/video.gif" width="480" controls></video>
17         <p id="alerta" style="background-color: red; color: white;"></p>
18     </body>
19 </html>

```

El siguiente paso es adaptar el contenido del archivo JS. Vamos a editarlo para agregarle en la parte superior la declaración de dos nuevas variables: **hconc** y **video**:

```

...
var video = document.getElementById("video");
var hconc = navigator.hardwareConcurrency;

```

Con esto, incluimos en la variable **video** la etiqueta HTML correspondiente al mismo, y en la variable **hconc** almacenamos el valor devuelto por **navigator.hardwareConcurrency**.

Al final de la sentencia **Switch** utilizada para cargar la imagen más óptima añadimos una serie de **If** que analizarán la combinación de **hconc** y **tconn**. Con el resultado obtenido, reconfiguraremos la variable **hconc** con un valor acorde al video que deberá mostrar:

```

if (hconc >= "6" && tconn >= "3g") {hconc = "6"}
if (hconc === "4" && tconn >= "3g") {hconc = "4"}
if (hconc === "4" && tconn < "3g") {hconc = "2"}
if (hconc === "2" && tconn >= "2g") {hconc = "2"}
if (hconc === "2" && tconn == "2g") {hconc = "1"}
if (hconc === "1" && tconn > "2g") {hconc = "1"}
if (hconc === "1" && tconn == "2g") {hconc = "0"}

```

Vaciamos a continuación el elemento **<p>**, cuyo id es **"alerta"**:

```
document.getElementById("alerta").innerText = "";
```

Ahora analizamos el valor de **hconc**. Si es mayor que 1, cargamos la portada del video; y si es menor, cargamos una portada genérica:

```

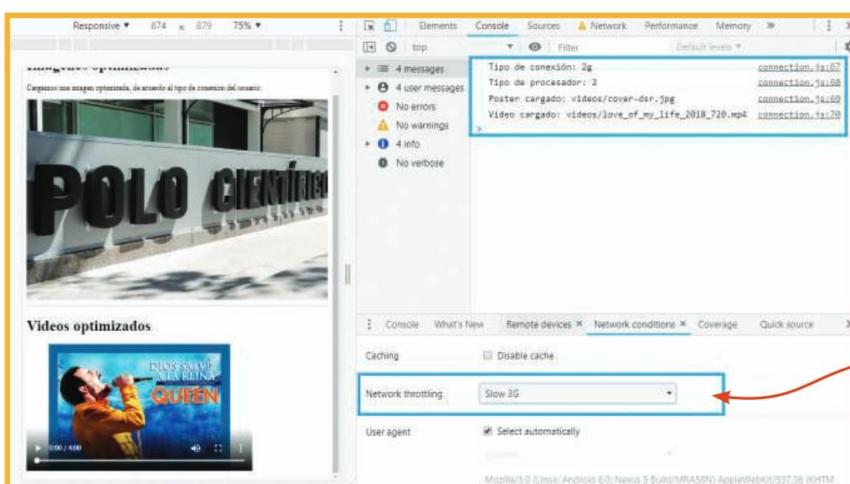
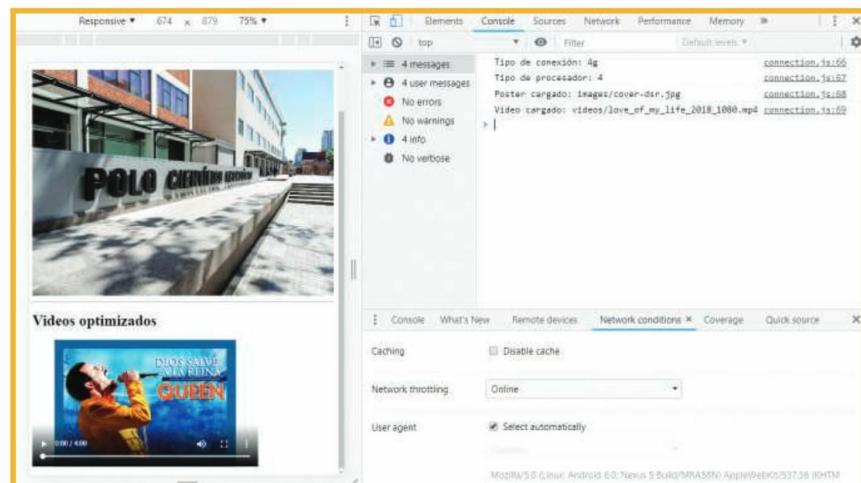
if (hconc > "1") {
    video.setAttribute("poster", "images/cover-dsr.jpg");
} else {
    video.setAttribute("poster", "images/video.gif");
}

```

Por último, agregamos una nueva sentencia **Switch**, que analiza el valor de la variable **hconc**, y con esto determina la calidad del video que se va a cargar:

```
switch (hconc) {  
    case ("6"):  
    case ("4"):  
        video.setAttribute("src", "videos/love_of_my_life_2018_1080.mp4");  
        break;  
    case ("2"):  
        video.setAttribute("src", "videos/love_of_my_life_2018_720.mp4");  
        break;  
    case ("1"):  
        video.setAttribute("src", "videos/love_of_my_life_2018_480.mp4");  
        break;  
    case ("0"):  
        document.getElementById("alerta").innerText = "Su dispositivo es muy  
lento para reproducir videos.";  
        video.setAttribute("src", "");  
        break;  
    default:  
        video.setAttribute("src", "videos/love_of_my_life_2018_480.mp4");  
        break;  
}  
}
```

Finalmente, ya estamos en condiciones de probar la calidad de video que se cargará en nuestra página web.



Podemos aprovechar la opción **Network Throttling** de Chrome para simular diferentes conexiones y así ver cómo se carga una calidad de video acorde.

Capturar fotografías y videos



Conozcamos las ventajas que propone JS para sacar provecho de las capacidades de hardware multimedia como webcams, cámaras fotográficas y micrófonos incluidos en computadoras y dispositivos móviles.

JavaScript, a través de la API `getUserMedia()`, propone acceder a la cámara fotográfica de dispositivos móviles y/o computadoras para realizar capturas de fotos o filmaciones, sin necesidad de recurrir

a un programa o app nativa. Este método estuvo originalmente incluido en **Navigator**. `getUserMedia()` y, luego, fue agrupado como subcaracterística de **Navigator**. **MediaDevices**.

navigator.MediaDevices

A través de **MediaDevices**, el objeto Navigator permite al desarrollador conocer todos los dispositivos físicos (hardware) de entrada/salida que tiene la computadora o dispositivo móvil del usuario que accede a una web. Para ver cómo trabaja, podemos escribir la siguiente sentencia en la línea de comandos de las Herramientas para el desarrollador:

```
navigator.mediaDevices
```

```
Elements Console Sources Network Performance Memory Application
Default levels ▾
top:
No messages
No user me...
No errors
No warnings
No info
No verbose
MediaCapabilities ()
navigator.mediaDevices
MediaDevices {ondevicechange: null}
ondevicechange: null
proto : MediaDevices
enumerateDevices: f enumerateDevices()
getSupportedConstraints: f getSupportedConstraints()
getUserMedia: f getUserMedia()
ondevicechange: {...}
constructor: f MediaDevices()
Symbol(Symbol.toStringTag): "MediaDevices"
get ondevicechange: f ondevicechange()
set ondevicechange: f ondevicechange()
proto : EventTarget
```

Obtendremos un array de datos y, al desplegar los subniveles, encontraremos una función denominada `enumerateDevices()`.

Si ejecutamos a continuación el comando completo, llegaremos al array de hardware que tiene el dispositivo móvil o la computadora:

```
navigator.mediaDevices.enumerateDevices()
```

Si otra vez desplegamos el array para ver todo su contenido, podremos identificar si el equipo en cuestión tiene cámara o no. Veamos en la siguiente imagen cómo se identifica el dispositivo vinculado a una webcam:

```
navigator.mediaDevices.enumerateDevices()
Promise {<pending>}
proto : Promise
[[PromiseStatus]]: "resolved"
[[PromiseValue]]: Array(8)
0: InputDeviceInfo {deviceId: "default", kind: "audioinput", label: "Predeterminado - Micrófono"}
1: InputDeviceInfo {deviceId: "communications", kind: "audioinput", label: "Comunicaciones - Micrófono"}
2: InputDeviceInfo {deviceId: "96c2aa9303f902639200b08a0cc6027875f2e2dccc50d175387a8e339b87632", kind: "videoinput", label: "HD Webcam (1bcf:2c17)"}
3: InputDeviceInfo {deviceId: "504b9d9c34df7bf99748703e0a334c922c4cd8016af1cebd9a800745e888490", kind: "audioinput", label: "Predeterminado - Altavoces (Dispositivo de High Definition Audio)"}
4: InputDeviceInfo {deviceId: "c0b777be560329eab9d09fabb6b499e1531ab9d071474488cfddaa6b5ccff6ded5", groupId: "652fb4e2df8dcc15eee1955cf2e929fe08357843d5bc033cbff6dd3843f4ca", kind: "videoinput", label: "HD Webcam (1bcf:2c17)"}
5: MediaDeviceInfo {deviceId: "default", groupId: "f82f5de6785eb1455b9f92f7ea4ba05f03c75b437aa8c6c08b9d6e15f4cef00c", kind: "audiooutput", label: "Predeterminado - Altavoces (Dispositivo de High Definition Audio)"}
```

La integración de Promises

En estos últimos años, se comenzó a hablar mucho de **Promises** dentro del ecosistema JavaScript. Lo que propone esto es un objeto que representa la eventual finalización o falla durante la ejecución de un proceso determinado utilizando una operación asincrónica. Antes había que pensar en crear dos funciones completamente distintas para un determinado

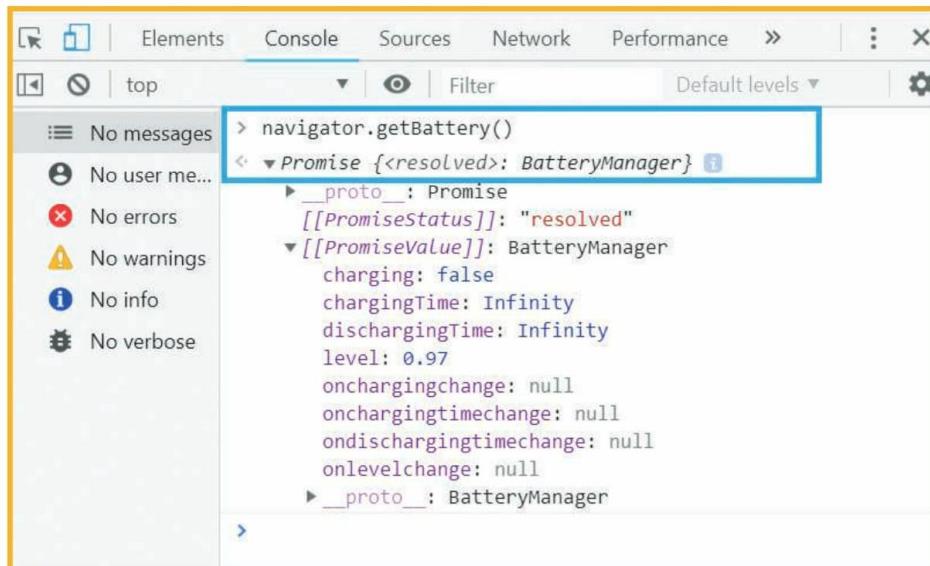
proceso: una abocada a ejecutar el proceso en sí y devolver un posible resultado al resto del algoritmo que se ejecutaba; y otra que “analizaba” si el algoritmo no devolvía un objeto, para informarle al programa o al usuario que la operación había fallado. Veamos a continuación un ejemplo de cómo se estructuraba antes una operación asincrónica:

```
// Función de ejecución
function successCallback(result) {
    console.log("El archivo de audio está completo en la URL: " + result);
// Función de falla
function failureCallback(error) {
    console.log("Se ha generado un error al crear el archivo de audio:
" + error);
// Función asincrónica con sus parámetros de control
createAudioFileAsync(audioSettings, successCallback, failureCallback);
```

En la actualidad, con la integración de **Promises**, el código se resuelve del siguiente modo:

```
createAudioFileAsync(audioSettings).then(successCallback, failureCallback);
```

De esta forma, integrar **Promises** en nuestro código nos permitirá manipular más ágilmente los acontecimientos asincrónicos, sin tener que utilizar, por ejemplo, contadores del tipo **setTimeout()** para esperar un período determinado antes de ejecutar un **callBack**.



También debemos tener presente que Promises permite ejecutar un conjunto de funciones asincrónicas consecutivas, invocando **.then** de manera anidada, y por supuesto, también integrar la función **Catch()** para el tratamiento de errores.

```
hazAlgo()
  .then(resultadoo => hazAlgoMas(resultado))
  .then(nuevoretulado => hazUnaTercerCosa(nuevoResultado))
  .then(resultadoFinal => console.log("El resultado final es: ${resultadoFinal}"))
  .catch(errorEnResultados);
```

getUserMedia()

Regresemos al objetivo de este proyecto: capturar fotografías con la cámara del dispositivo o computadora del usuario. Para esto, utilizaremos el objeto **Navigator**, el de los ejercicios anteriores. Como bien dijimos, este objeto incluye la propiedad

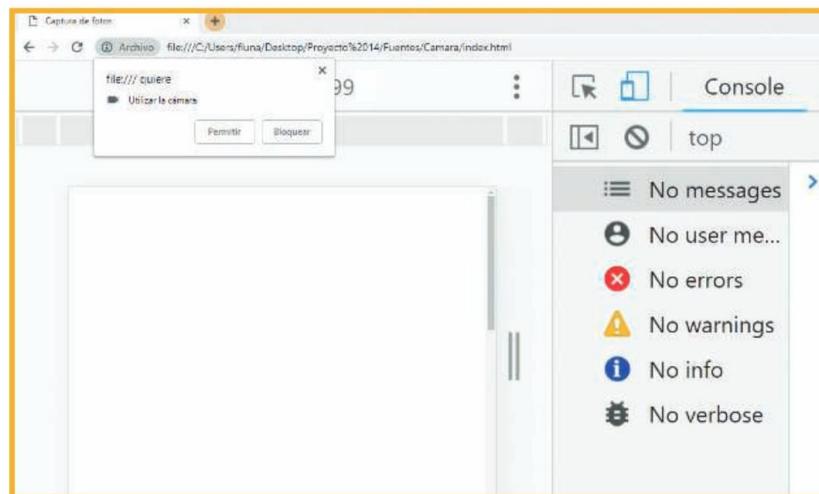
mediaDevices, que accede a todo el hardware de la computadora o dispositivo que carga nuestra página web. Esta, a su vez, contiene un método denominado **getUserMedia()**, que nos permitirá acceder al hardware del usuario. Entonces, ¿cómo lo invocamos?:

```
navigator.mediaDevices.getUserMedia(params);
```

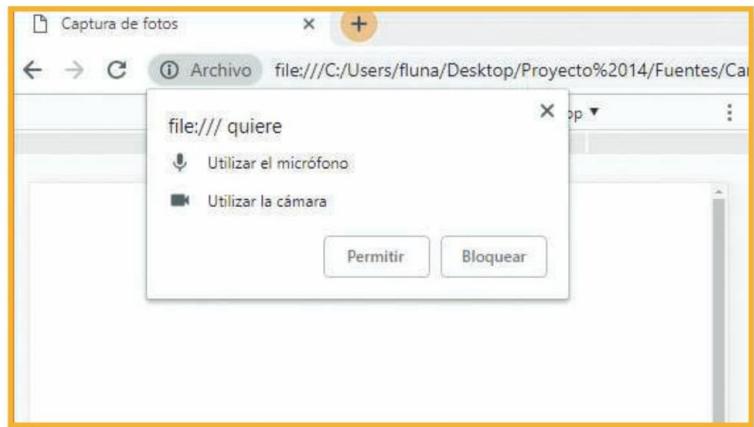
Como vemos en el ejemplo de código anterior, este método requiere de parámetros, que en este caso, sirven de configuración. Por lo general, la multimedia a la cual accedemos en cualquier hardware moderno está compuesta por **Audio** y **Video**. Y para utilizar uno, otro o ambos, debemos indicar cuáles son los que estamos invocando. Esto se realiza de la siguiente forma:

```
navigator.mediaDevices.getUserMedia({
  video: true,
  audio: false
})
```

Como podemos ver, en nuestro caso estamos por utilizar solo el objeto **video**, sin **audio**. Al invocar este método, lo que hacemos, tanto en móviles como en computadoras (y por cuestiones de seguridad), es pedirle permiso al usuario para utilizar uno o ambos dispositivos de hardware de su equipo.



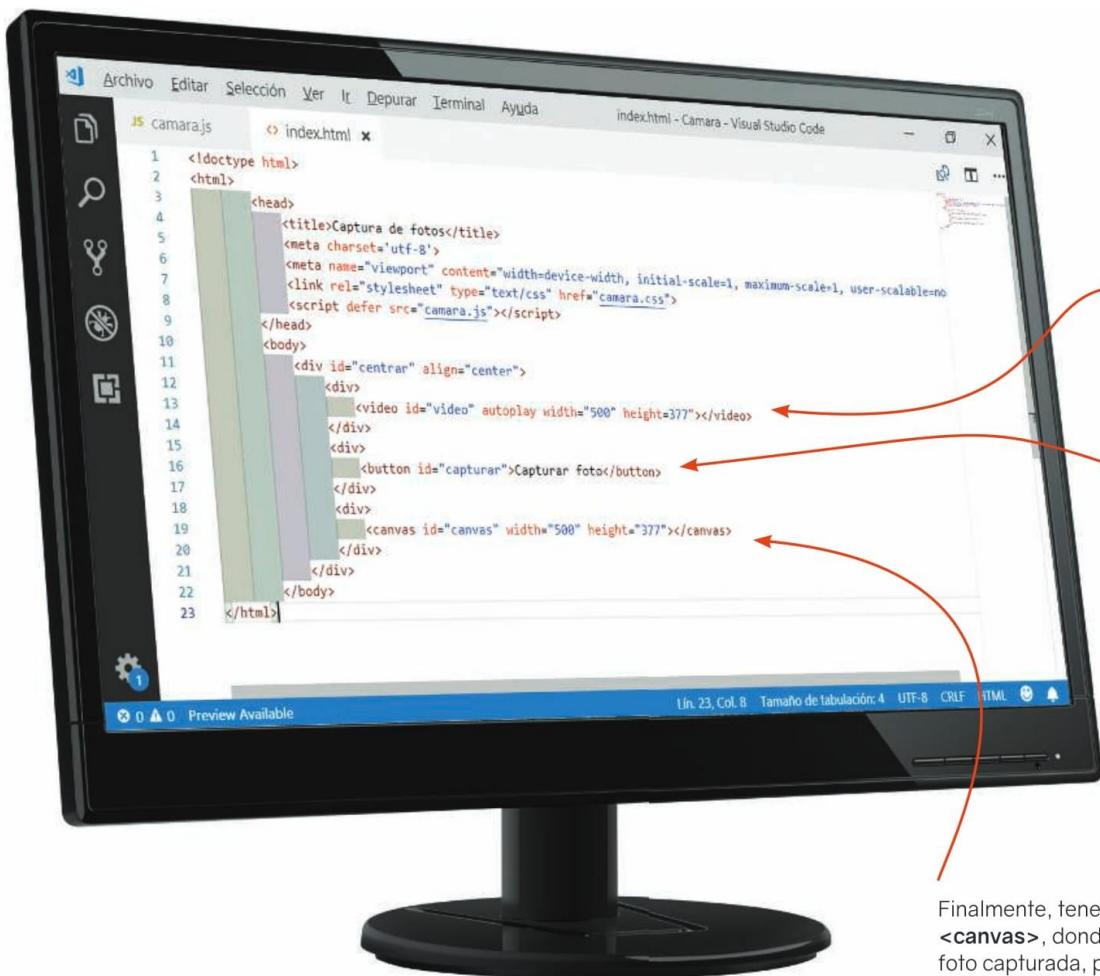
De esta forma se invoca, por ejemplo, el mensaje de permisos de uso de recursos de hardware que solemos ver en el extremo superior izquierdo de la barra de navegación. Si le asignamos el valor **true** al parámetro **audio** anterior, entonces la solicitud de permiso variará como se observa en la imagen.



Concepto y uso de la API Stream

Tanto **Media Capture** (que veremos a continuación) como **Streams API** son APIs relacionadas a **WebRTC**, que provee soporte para la transmisión de audio y video. A través de ellas, accedemos a las interfaces y métodos necesarios que nos permiten trabajar un **stream**. Podemos definir el stream como un flujo de audio y/o video

que parte de cero o más objetos del tipo **track** anidados, los cuales terminan conformando el flujo multimedia mencionado. Para entender técnicamente cómo funciona esto, descargamos el proyecto **Proyecto-Camara-base.zip** y creamos a continuación uno nuevo en **VSCode**, utilizando estos archivos base.



El HTML de este proyecto contiene tres componentes. El primero es el denominado tag **<video>**, que se ocupará de recibir el streaming que generemos.

El segundo es un **<button>** que disparará la captura de la foto del streaming recibido.

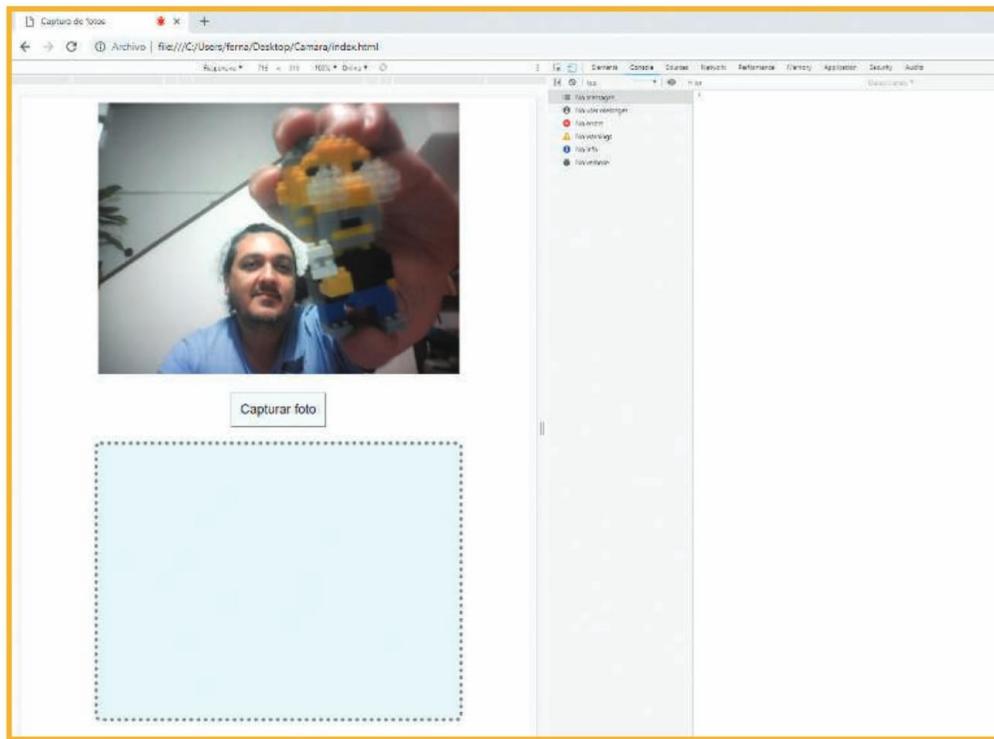
Finalmente, tenemos un tag **<canvas>**, donde almacenaremos la foto capturada, para luego guardarla en el equipo.

Codificar nuestro JS

Abrimos el archivo JS asociado a este proyecto y, dentro del método **AddEventListener** creado, comenzamos a escribir el código. Lo primero que hacemos es agregar el llamado a la API **getUserMedia()**, de la cual solo usaremos el video:

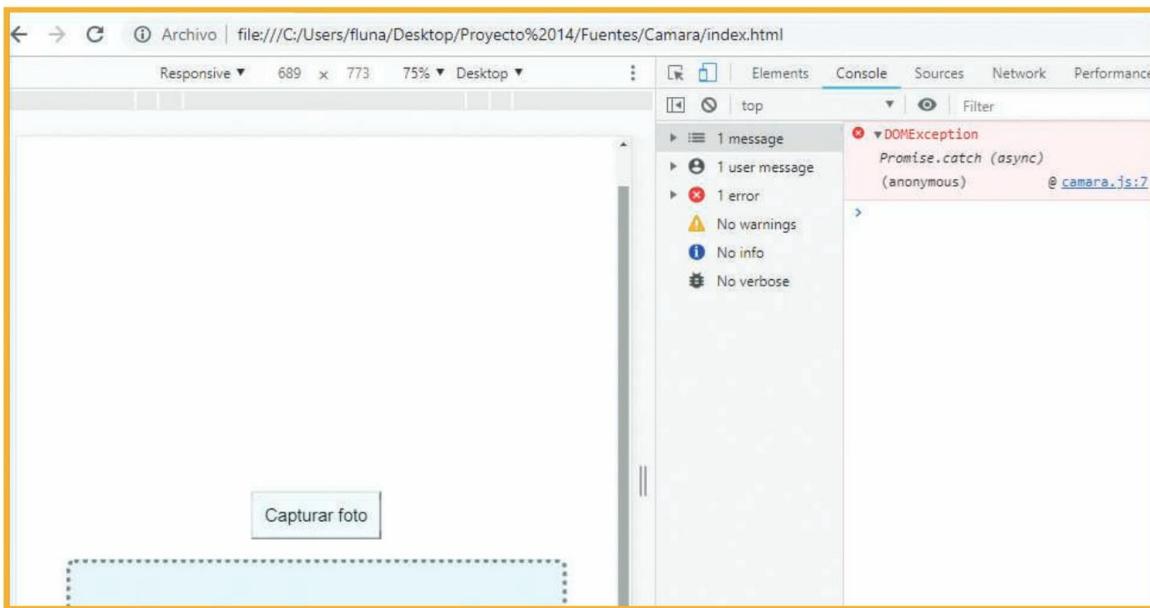
```
document.addEventListener("DOMContentLoaded", function() {
    navigator.mediaDevices.getUserMedia({
        video: true,
        audio: false
    })
    .then(stream => video.srcObject = stream)
    .catch(console.error);
...
});
```

Aquí utilizamos el objeto **stream**, el cual llegará mediante una operación asincrónica que espera la confirmación del usuario, quien debe aceptar transmitir video desde su dispositivo. Si hay un error, será capturado y mostrado en la consola, a través de **.catch**.



Promise solucionó y ordenó el cómputo asincrónico, que aún es crucial en este tipo de desarrollo de software, dado sus tiempos de carga, respuesta, o disponibilidad técnica en el navegador web.

En caso de que el usuario no acepte la transmisión del video, el tag <video> del documento HTML quedará vacío. No se visualizará nada en él, mientras que la consola de JavaScript incluida en las Herramientas para el desarrollador de Chrome arrojará una excepción (o error) en el DOM.



A continuación, agregamos el código que se ejecutará al momento de hacer clic o tap sobre el componente HTML <button>:

```
document.getElementById("capturar").addEventListener("click", function() {
    canvas.getContext("2d").drawImage(video, 0, 0, 500, 377);
    ...
})
```

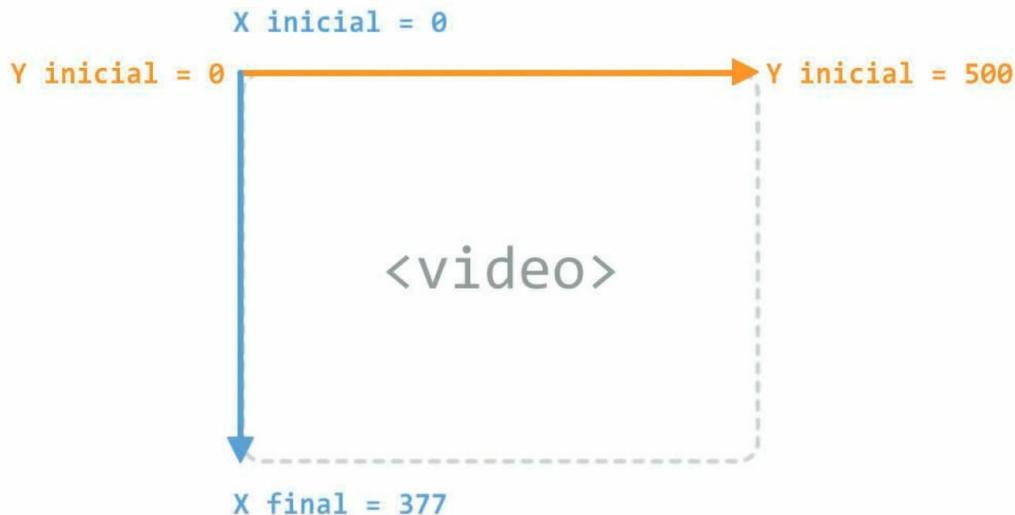
Capturamos el evento clic del botón. Cuando este se ejecuta, mediante el método **drawImage()**, dibujamos contenido en el objeto **canvas**, uno de los **frames** que es transmitido en el componente <video>. Esto es el equivalente a tomar una fotografía. Antes de la invocación de este método, debemos definir al objeto **canvas**, que en el contexto actual que

se representará dentro de sí, es un contexto **2d** (dos dimensiones). Los parámetros que recibe el método **drawImage()** corresponden a las coordenadas que debemos dibujar (X inicial = 0, Y inicial = 0, X final = 377, Y final = 500) del streaming de video. Estas coordenadas equivalen a la dimensión en la que actualmente se reproduce el video.

Canvas está presente en HTML como un elemento y en JavaScript como un objeto. Nos permite dibujar gráficos, visualizar imágenes, crear animaciones o procesar el renderizado de video en tiempo real.

Podemos ver cómo se reparten las coordenadas que debemos pasar como parámetros al método **drawImage()**. Este recibe cinco parámetros en total, siendo el primero de ellos el objeto desde donde se realizará la captura de imagen (video).

canvas.getContext("2d").drawImage(video, 0, 0, 500, 377);



Almacenar la foto en el equipo

El último paso es almacenar la foto en la computadora o dispositivo móvil del usuario. Para esto, a continuación del método **drawImage()**, agregamos el siguiente código:

```
canvas.setAttribute("src", canvas.toDataURL('image/png'));
guardarFoto();
```

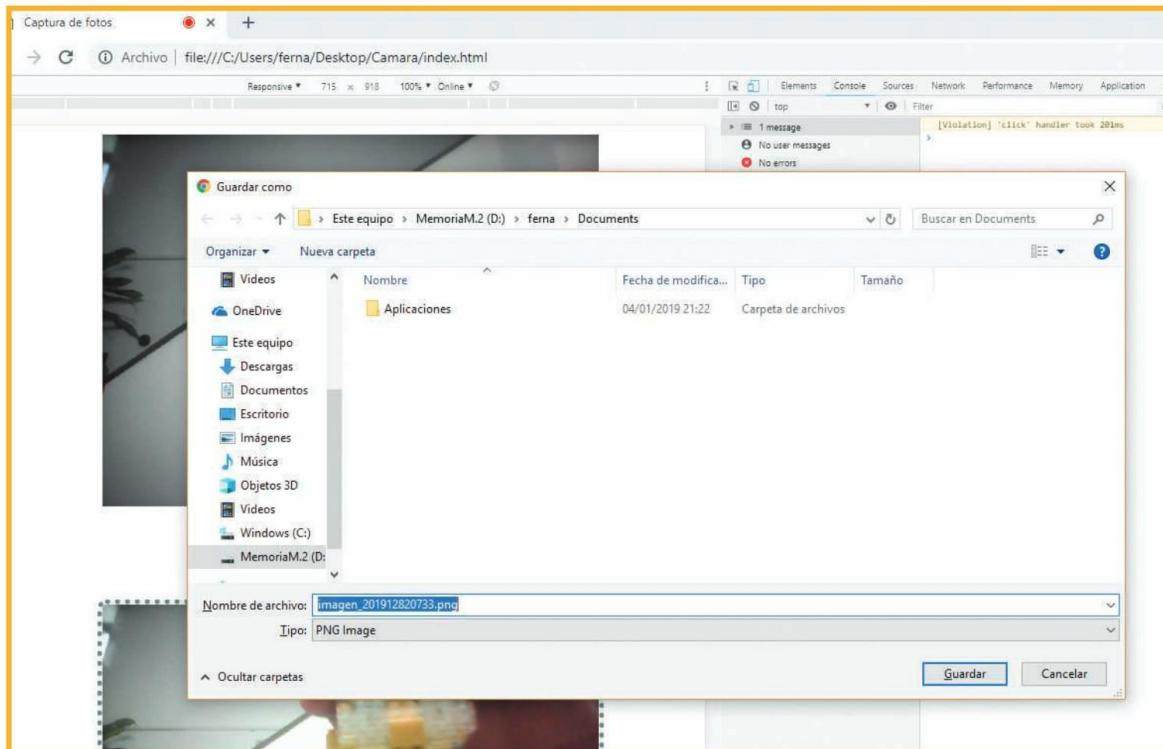
Utilizamos el atributo **src** del componente **canvas**, para convertir el **frame** capturado en una imagen. Luego, mediante el método **toDataURL()** del componente **canvas**, le indicamos que dicho objeto es del tipo **image/png**. De esta forma, definimos que descargaremos una imagen al equipo o dispositivo, a través de la función **guardarFoto()**. Creemos dentro del mismo **AddEventListener**, la función mencionada:

```
function guardarFoto() {
    var link = document.createElement("a");
    link.download = "imagen_" + obtenerFecha() + ".png";
    link.href = document.getElementById("canvas").toDataURL()
    link.click();
}
```

Básicamente, creamos la variable **link**, del tipo hipervínculo, a la cual le asignamos en su propiedad **download** un nombre de imagen, que obtenemos concatenando “**imagen_**” junto con la función **obtenerFecha()**, que generaremos a continuación. Finalmente, el vínculo de referencia de esta imagen es tomado desde la propiedad **toDataURL()** del elemento **canvas** y, por último, simulamos un clic para que se genere la descarga de la imagen:

```
function obtenerFecha() {  
    d = new Date();  
    fh = d.getFullYear() + ' ' + (d.getMonth() + 1) + ' ' +  
    d.getUTCDate() + ' ' + d.getHours() + ' ' + d.getMinutes() + ' ' +  
    d.getSeconds();  
    return fh;  
}
```

La función **obtenerFecha()** crea un objeto del tipo **Date()**, y almacena en una variable **fh** la concatenación de los datos Año, Mes, Día, Hora, Minutos y Segundos. Por último, retorna esta información a través del mismo nombre de función. De esta manera, para cada imagen que se va a descargar se concatena un nombre único, de modo que no sobrescriba una anterior.



Incluir la fecha y hora del instante en que descargamos una imagen nos permitirá evitar la sobre escritura de archivos y el renombrado indexado que muchas veces pueden confundir al usuario final.

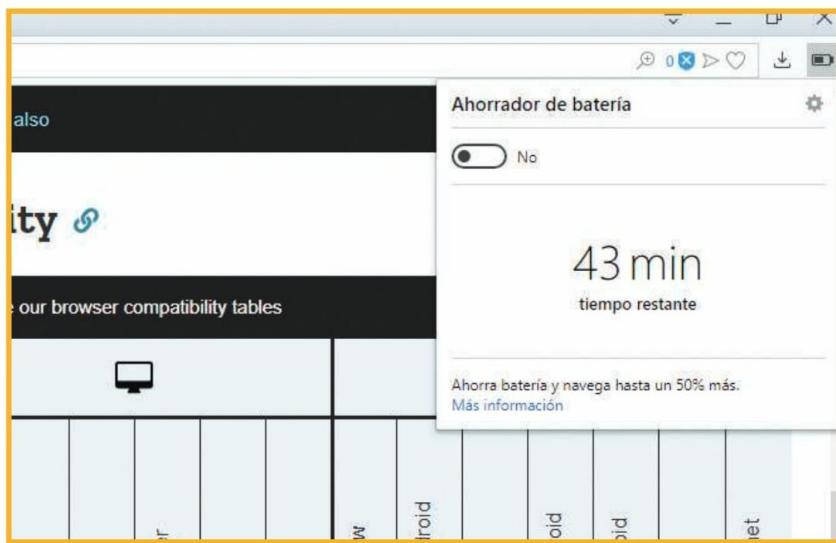
Acceder al hardware de los dispositivos



Comenzaremos a explorar en profundidad el hardware de los dispositivos del usuario conociendo, en esta primera instancia, el consumo de batería de los dispositivos móviles y cómo generar notificaciones mediante vibración.

Continuando en el terreno experimental que propone JavaScript, una opción muy vista este último tiempo dentro de los navegadores web es la de poder conocer el nivel de batería de los dispositivos del usuario. Si hemos prestado atención a las últimas implementaciones de Google Chrome en los dispositivos móviles y

computadoras portátiles, este navegador ha incluido una función que notifica en pantalla cuando un equipo funciona con batería y, además de ofrecernos la posibilidad de ahorrar datos durante la navegación, nos avisa el tiempo estimado de batería que nos queda.

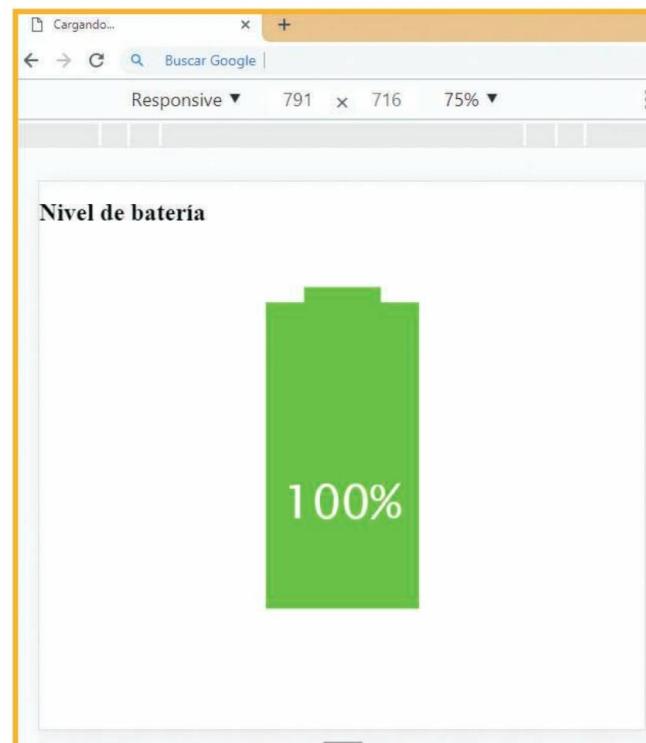


Como desarrolladores, no está mal incluir el análisis de este tipo de dato, ya que la web de hoy pasa, en gran parte, por el consumo desde pantallas móviles; entendiendo por móviles no solo tablets y smartphones, sino también notebooks y netbooks, que dependen en gran medida de una batería para su correcto funcionamiento.



navigator.getBattery

El objeto **Navigator** suma a su cúmulo de características la posibilidad de medir el nivel de batería de los dispositivos donde se ejecuta una web, a través del método **getBattery()**. De esta forma, como desarrolladores que acceden a este dato, podemos modelar procesos de carga de una web evaluando el nivel de batería, para mostrar una mejor imagen, o habilitar o deshabilitar la posibilidad de cargar y/o reproducir un video de alta calidad.



Veamos el conjunto de características que obtenemos al utilizar este método de análisis de batería de los equipos, para finalmente integrarlo a un proyecto funcional que nos permita entender en la práctica cómo se comporta **getBattery()**.

De acuerdo con **Mozilla Developer Network**, la API de especificación **Battery Status** se encuentra en una fase de recomendación. Actualmente está implementada en **Google**

Chrome, Opera y Microsoft Edge, por el lado de los navegadores de escritorio.

En el terreno mobile, solo está presente en Chrome y Opera para Android, Samsung Internet Browser y Android WebView. Esto último nos garantiza el poder contar con esta característica si desarrollamos soluciones basadas en **PWA, Apache Cordova, Ionic y Phonegap**.

Browser compatibility

Take this quick survey to help us improve our browser compatibility tables													
	Desktop						Mobile						
	Chrome	Edge	Firefox	Internet Explorer	Opera	Safari	Android webview	Chrome for Android	Edge Mobile	Firefox for Android	Opera for Android	Safari on iOS	Samsung Internet
Basic support	👎	38	?	16 — 52	No	25	No	40	38 ★ —	No	10 — 16 -x-	25	? — ?

Cómo invocar el método getBattery()

La integración del método **getBattery()** se realiza también utilizando **Promises**, tal como vemos en el siguiente código:

```
navigator.getBattery().then(function(battery) {
    // Obtengo valores y ejecuto funciones
})
```

Si se ejecuta correctamente el método (porque está disponible en el motor de navegación), entonces se ejecuta la función **function(battery)**, donde el parámetro indicado a la función, **battery**, se transforma en un objeto para proveer diferente información del sistema de hardware. Así, podremos tomar diferentes decisiones en cuanto al comportamiento de la web. Veamos en la siguiente tabla qué podemos consultar a este objeto battery:

PROPIEDAD	VALOR	DESCRIPCIÓN
level	Decimal (entre 1 y 0)	Devuelve un valor decimal entre 1 y 0, que debemos multiplicar por 100 para obtener el porcentaje de batería actual.
charging	Boolean	Indica si el equipo se encuentra o no conectado a un cargador.
chargingTime	Integer	Si el equipo está conectado a un cargador, a través de esta propiedad tendremos un estimado (en segundos) del tiempo restante de carga.
dischargingTime	Integer	Si no está conectado a un cargador, esta propiedad devuelve un estimado (en segundos) del tiempo restante de batería.

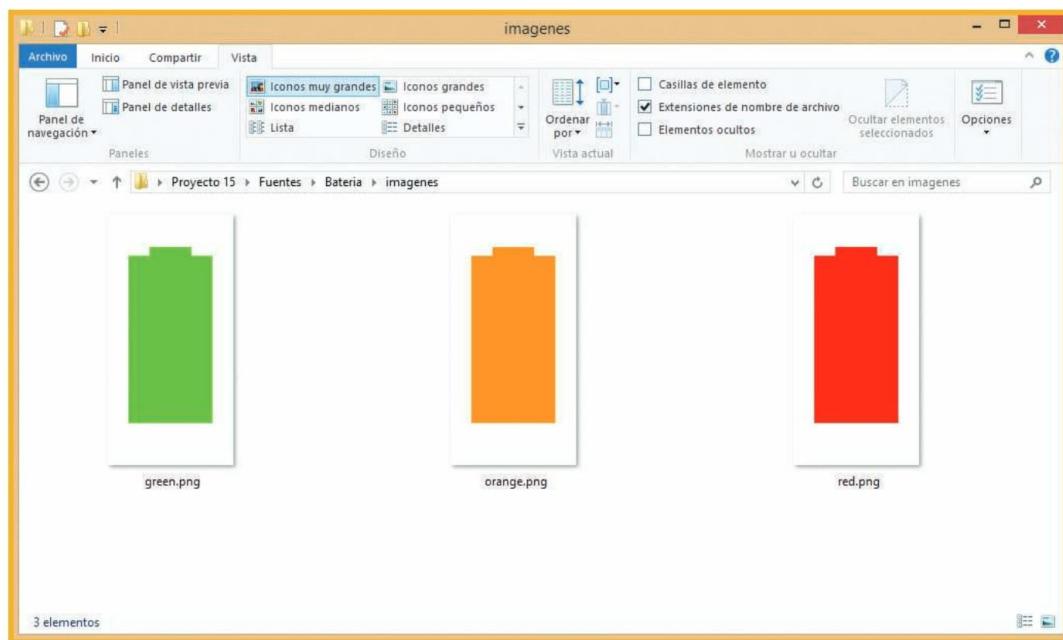
EVENTO	DESCRIPCIÓN
levelchange	Event Listener que nos permite detectar el momento en el cual cambia el nivel de la batería (aumenta o disminuye).
chargingTimeChange	Event Listener que detecta cuando cambia el valor del tiempo de carga (cuando este aumenta).
dischargingTimeChange	Event Listener que detecta cuando cambia el valor del tiempo de descarga (cuando este disminuye).

Con el buen uso de este método podemos definir la carga progresiva de imágenes, mostrar imágenes con mayor o menor calidad, reproducir automáticamente videos o limitar la resolución del video, entre otras opciones útiles para una aplicación web exitosa.

/<Ejercicio práctico>

Medidor de batería

Pongamos manos a la obra con nuestro próximo proyecto práctico. Construiremos una aplicación web que mida el nivel de batería de un dispositivo, actualizando su valor en pantalla y alertando, mediante la consola, cuando llega a determinados niveles. Para empezar, descargamos el archivo **Proyecto-nivel-bateria-base.zip**.



Este proyecto web cuenta con una subcarpeta denominada **imágenes**, la cual contiene tres gráficos en forma de batería, titulados **green.png**, **orange.png** y **red.png**. Nuestro propósito será, por un lado, mostrar el nivel de batería del equipo donde se ejecuta esta página web y, cuando dicho nivel difiera de determinados valores, cambiar la imagen por la de un indicativo de color que notifique visualmente el nivel correspondiente.

Creamos un nuevo proyecto en Visual Studio Code, agregamos el contenido de este archivo .ZIP y editamos a continuación el archivo JS. Encontraremos en él el siguiente código:

```
var i = document.body;
var nivel;
document.addEventListener("DOMContentLoaded", function() {
    bateria();
});
```

Tenemos dos variables declaradas: **i** y **nivel**. La primera almacena el valor del cuerpo del documento, la cual utilizaremos más adelante para cambiar la imagen de color por la otra imagen que corresponda. La segunda será la que almacenará el nivel de batería que leeremos. Dentro del Event Listener que detecta la finalización de carga de la página, tenemos declarada la función **bateria()**.

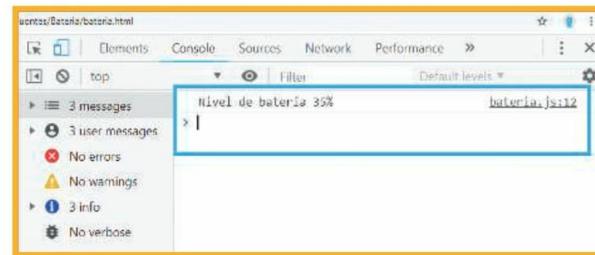
Escribamos a continuación el código para dicha función:

```
function bateria() {
    navigator.getBattery().then(function(battery) {
        ...
    });
}
```

Como podemos ver en este código, invocamos al método `getBattery()` utilizando **Promises**. Si puede ejecutarse en el navegador, entonces activamos un **Event Listener**, que escuchará cuando se produzca un cambio en el nivel de batería del equipo:

```
battery.addEventListener("levelchange", function() {
    nivel = (battery.level * 100).toFixed(0);
    document.getElementById("pbat").textContent = nivel + "%";
    ...
})
```

Utilizamos la variable **nivel**, donde almacenamos el valor devuelto por **battery.level**, multiplicándolo por **100** y redondeándolo para conseguir así un número entero para mostrar. Luego, en el tag **<p>** dentro de la imagen, escribimos el valor de batería obtenido, expresado como porcentaje. A esta altura, ya podemos ejecutar nuestro proyecto en una computadora o dispositivo que cuente con batería, y obtener el nivel de carga.



A continuación, creamos la función que actualizará la imagen de fondo, acorde a determinados niveles de batería. También escribimos en el título de la pestaña de navegación si la batería se está cargando o descargando. Para esto, agregamos lo siguiente dentro del bloque de código anterior:

```
...
actualizoImagenNivel(nivel);
if (battery.charging) {document.title = "Cargando...";}
else {document.title = "Descargando...";}
...
```

Consultamos la propiedad **charging**, que devuelve un booleano. Acorde a este, cambiamos el título de la pestaña de navegación, donde indicamos si la batería se está cargando o descargando, utilizando **title** del objeto **document**. Luego, creamos la función **actualizoImagenNivel()**:

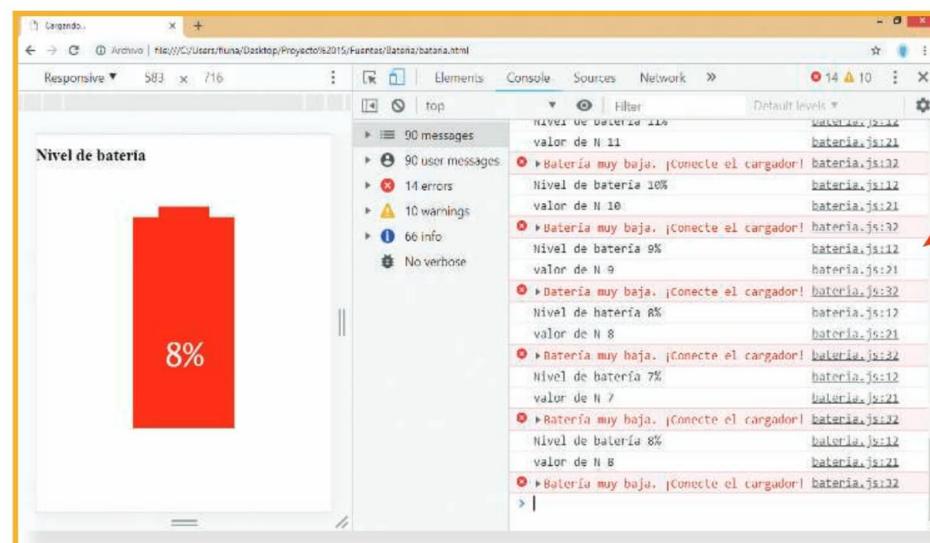
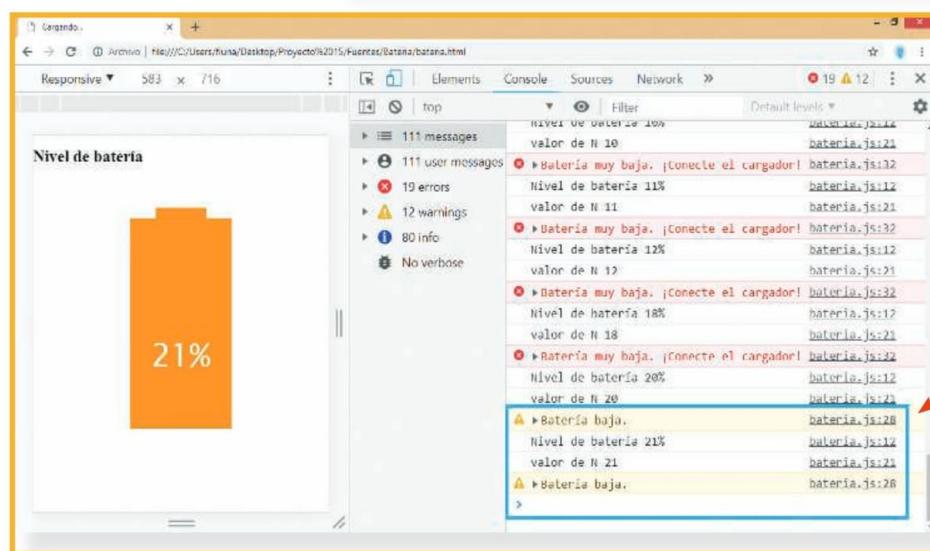
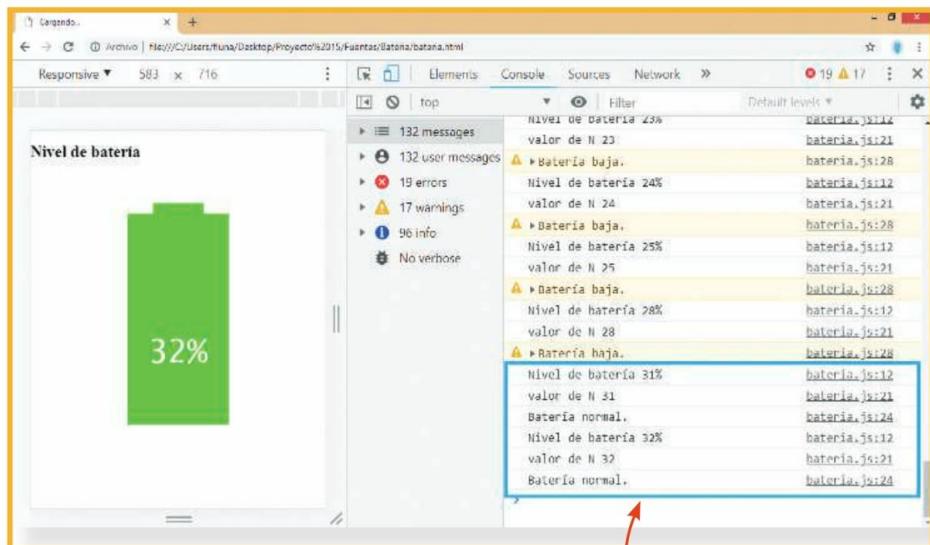
```
function actualizoImagenNivel(n) {
    console.log("valor de N " + n);
    if (n <= 100 && n >= 30) {i.style.backgroundImage = "url(imagenes/green.png)";
        console.info("Batería normal.");}
    if (n <= 29 && n >= 20) {i.style.backgroundImage = "url(imagenes/orange.png)";
        console.warn("Batería baja.");}
    if (n <= 19) {i.style.backgroundImage = "url(imagenes/red.png)";
        console.error("Batería muy baja. ¡Conecte el cargador!");}
}
```

En este código analizamos el valor **n**, pasado como parámetro al invocar esta función, el cual contiene el nivel de batería obtenido anteriormente. Con este dato, creamos una serie de sentencias **If**, las cuales nos permiten analizar el nivel de batería y, si está dentro de determinados parámetros, mostrar una u otra imagen.

También utilizamos **console.info()**, **console.warn()** y **console.error()** para notificar o alertar mediante la consola sobre los diferentes niveles de batería.

Adaptar el gráfico al nivel de batería

Como hemos visto en el código anterior, el gráfico de fondo que simula una batería cambia de acuerdo con los diferentes niveles que la funcionalidad de esta API va reportando.



Cuando el nivel de batería se mantiene entre 100% y 30%, la imagen de fondo es de color verde. Por cada evento que lee el nivel de batería, ejecutamos a su vez `console.info()`, para indicar su estado óptimo.

Cuando el nivel de batería desciende por debajo del 30%, y se mantiene igual o por arriba del 20%, la imagen de fondo cambia por la de color naranja, y en la consola del navegador ejecutamos el evento `console.warn()`.

Por último, cuando el nivel de batería alcanza 19% o menos, la imagen cambiará por la de color rojo, y en la consola aparecerá una alerta de conexión del cargador, utilizando para ello `console.error()`.

navigator.vibrate()

Los equipos más modernos, usualmente incluidos bajo el paradigma móvil, suelen incluir un sensor de vibración para notificar al usuario ante determinados eventos. JavaScript engloba bajo el objeto **navigator** el método **vibrate()**.

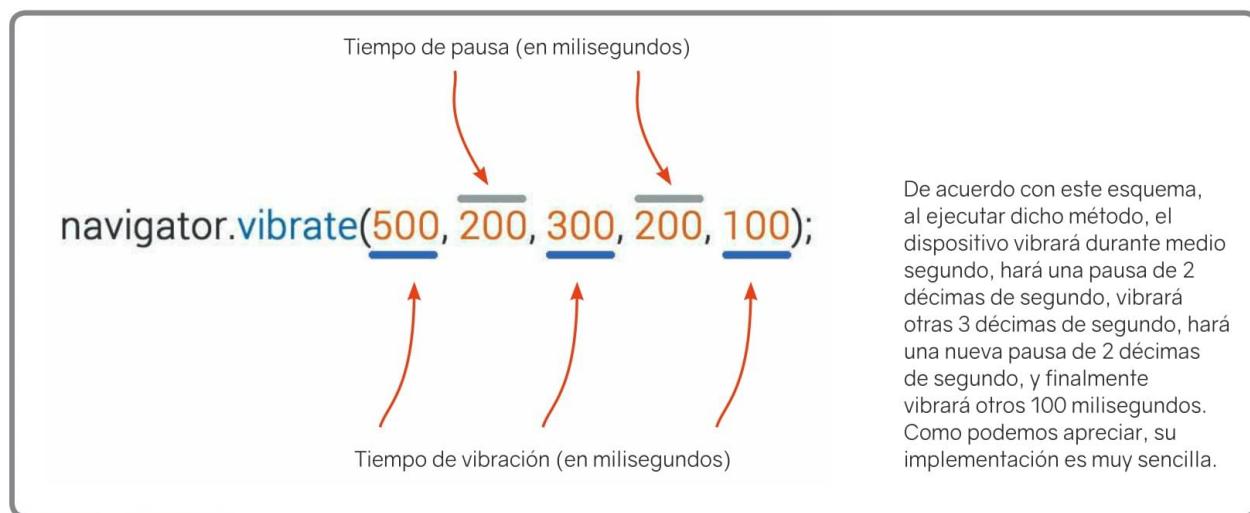
que permite producir este evento en dichos dispositivos. Su función es muy simple: se invoca la sentencia **navigator.vibrate(ms)**, pasándole los milisegundos que debe durar dicho efecto. Por ejemplo, si ejecutamos el comando:

```
navigator.vibrate(500);
```

El dispositivo móvil vibrará durante medio segundo.

Emitir pulsos continuos

El parámetro que recibe el método **vibrate()** no es único y fijo. Podemos especificarle una serie de parámetros consecutivos para generar una vibración pausada en el dispositivo. Veamos cómo hacerlo a través del siguiente diagrama:



Proyecto de ejemplo

Para probar esta característica, debemos descargar el proyecto de ejemplo **Proyecto-Vibracion-completo.zip**. Lo ejecutamos en una tablet, smartphone o computadora que cuente con esta característica. Para conseguir esto último, los invitamos a descargar un servidor proxy que permita poner este proyecto en línea, dentro de una red hogareña, o subir el contenido a algún sitio web desde donde podamos ejecutarlo a posteriori, en el dispositivo correspondiente.





RU RedUSERS PREMIUM

- ✓ Cientos de publicaciones USERS por una mínima cuota mensual.
- ✓ Siempre, donde vayas. On Line - Off Line. En cualquier dispositivo.
- ✓ Al menos 1 novedad semanal. Son 680 publicaciones y sumando...!!
- ✓ Incluye: eBooks - Informes USERS - Guías USERS - Revistas USERS y Power – CURSOS

SUSCRÍBETE

 usershop.redusers.com
 +54-11-4110-8700
 usershop@redusers.com

Los sensores de movimiento



Ya sea para aplicar en videojuegos, como para orientar nuestros puntos cardinales, los sensores de movimiento incluidos en los dispositivos móviles son de gran utilidad. Veamos cómo acceder a ellos utilizando nuestro fabuloso lenguaje JavaScript.

La integración de sensores de movimiento y ejes en tablet y smartphone ha generado un cambio beneficioso en los dispositivos móviles. Esto ha permitido, por ejemplo, desplegar una interacción

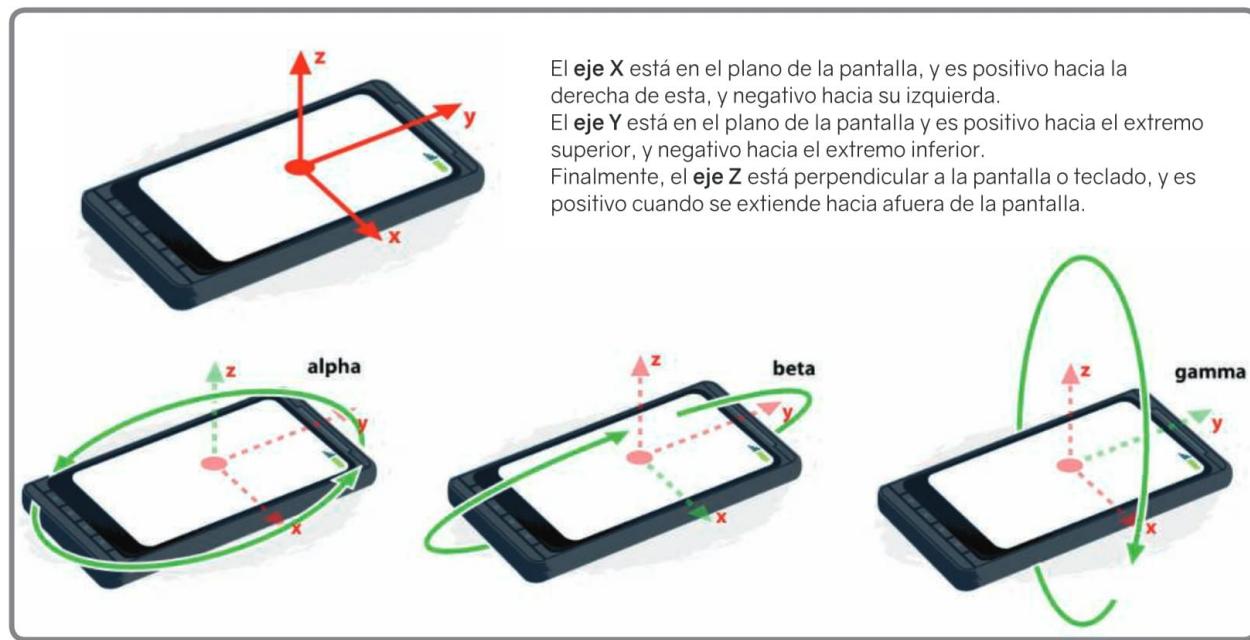
diferentes en los videojuegos, contar con una app brújula para guiarnos aún sin señal de antena, o utilizar un GPS que nos oriente hacia donde nos dirigimos en tiempo real.

Cómo procesar los eventos de orientación

En JavaScript, contamos con la posibilidad de procesar cualquier evento relacionado con la orientación de un dispositivo, utilizando precisamente el evento **deviceorientation**. Al ejecutar el código de detección de la orientación del dispositivo, lo que hacemos es “escuchar” el cambio de posición de los sensores incluidos en el acelerómetro. Su implementación es muy simple:

```
window.addEventListener('deviceorientation', function(e) {  
});
```

Ejecutando un **Event Listener**, donde invocamos el método **deviceorientation**, ya podemos comenzar a detectar cualquier cambio que se produzca en el dispositivo móvil. Este evento se encarga de reportar el valor de tres ejes principales: X, Y y Z.



Al alterar la posición de cualquiera de estos tres ejes, se está cambiando la graduación de los ángulos **Alpha**, **Beta** y **Gamma**. La orientación del dispositivo se determina sobre la base de cómo este se rota. Veamos de qué modo se comporta cada uno de los ejes:

ÁNGULO	COMPORTAMIENTO
Alpha	La rotación alrededor del eje Z causa que la rotación del ángulo Alpha cambie. El ángulo Alpha es 0° (cero grados) cuando el extremo superior del dispositivo apunta directamente hacia el eje Norte de la Tierra, e incrementa la forma en la que el dispositivo rota hacia la izquierda.
Beta	La rotación sobre el eje X se da cuando el dispositivo se aleja respecto a la posición del usuario, lo que hace que el ángulo Beta cambie. El ángulo Beta es 0° (cero grados) cuando el extremo superior e inferior del dispositivo están a la misma distancia del plano de piso, respecto de la superficie de la Tierra.
Gamma	La rotación sobre el eje Y indica que el dispositivo se mueve desde la izquierda hacia la derecha. Esto provoca que la rotación del ángulo Gamma cambie. El ángulo Gamma es 0° (cero grados) cuando los lados izquierdo y derecho del dispositivo están a la misma distancia del plano de piso, respecto a la superficie de la Tierra.

Implementaciones de la orientación y gravedad

A través de los sensores incluidos en los dispositivos móviles de gama media y alta, es posible no solo integrar una brújula en el equipo o interactuar con videojuegos, sino también detectar cuando un equipo está en caída libre y apagar a tiempo la energía del disco rígido, para evitar que los cabezales dañen los platos, como sucede en algunas unidades portátiles que utilizan los discos rígidos mecánicos.



La geolocalización basada en GPS no sólo permite medir las coordenadas Latitud y Longitud de un dispositivo con mucha precisión sino que también permite medir, en muchos casos, la distancia del dispositivo respecto

al plano de la tierra (Altitud). Estos detalles permiten calcular en algunas apps avanzadas, en qué piso se puede encontrar una persona, y hasta orientarla desde el software, por el interior de una vivienda o edificio.

Atención: Chrome, Firefox y los ejes

Cuando utilizamos el manejo de ejes y rotación de pantalla a través de JavaScript, debemos tener en cuenta que los navegadores Chrome y Firefox, y por ende los respectivos navegadores web que utilizan los motores de estos, tienen un manejo de ejes diferente. Por lo tanto, recomendamos que, al momento de desarrollar una solución que manipula los ejes y, en base a ellos, genera un determinado comportamiento de la aplicación, es preciso probarla en estos dos navegadores para controlar de manera correcta la forma en la cual responderá.



El objeto Math

Antes de proceder con el ejercicio práctico que concierne a este proyecto, vamos a analizar un poco qué nos ofrece JavaScript para el manejo de las operaciones matemáticas. JavaScript pone a nuestra disposición el objeto **Math**, a través del cual se invocan métodos y

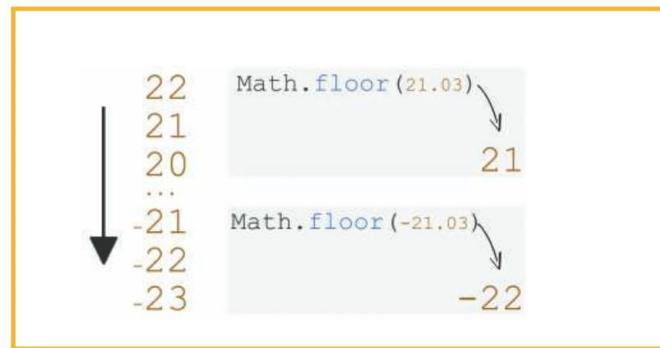
propiedades estáticas, que permiten operar fácilmente con cualquier tipo de número que necesitemos manipular.

Dentro del cúmulo de métodos de que disponemos en **Math**, podemos destacar los siguientes:

MÉTODO	DESCRIPCIÓN
<code>abs(x)</code>	Retorna el valor absoluto de un número.
<code>acos(x)</code>	Retorna el coseno de un arco.
<code>asin(x)</code>	Retorna el seno de un arco.
<code>atan(x)</code>	Retorna la tangente de un arco.
<code>ceil(x)</code>	Retorna el valor de un número redondeado hacia su número más próximo.
<code>cos(x)</code>	Retorna el coseno de un número en radianes.
<code>exp(x)</code>	Retorna el exponente de un número.
<code>floor(x)</code>	Retorna el número más próximo redondeando hacia el inferior más próximo.
<code>max(x, y, z...)</code>	Retorna el máximo número entre una serie de valores determinados.
<code>min(x, y, z...)</code>	Retorna el número mínimo entre una serie de valores determinados.
<code>random(x)</code>	Retorna un número al azar.
<code>round(x)</code>	Retorna un número redondeado hacia su entero más próximo.
<code>sqrt(x)</code>	Retorna la raíz cuadrada de un número.
<code>tan(x)</code>	Retorna la tangente de un ángulo.

Math.floor()

Para realizar nuestro próximo proyecto, utilizaremos el método **Math.floor()**, que nos devolverá el número inferior entero más próximo al valor que le pasemos como parámetro. En la siguiente imagen, podemos ver el comportamiento de esta función, dependiendo de si el número que le pasamos como parámetro es positivo o negativo.



/<Ejercicio práctico>

Rotar los ejes de una imagen

Para comprender mejor el uso del sensor acelerómetro de los dispositivos móviles a través de JavaScript, abordaremos a continuación un proyecto que cargará una imagen en dos dimensiones, y la rotará de acuerdo con el movimiento que realice el usuario con el dispositivo

móvil. Para empezar, descargamos el archivo **Proyecto-Ejes-Base.zip**. Descomprimimos la carpeta correspondiente y creamos un nuevo proyecto en **Visual Studio Code**. Al editarlo, veremos un archivo HTML base, que muestra alineada una imagen determinada.



Por el momento, este proyecto no realiza ninguna funcionalidad en particular. La misma debemos agregarla nosotros a través del archivo .JS relacionado. Para ello, editamos el documento HTML, y pulsamos **CTRL + clic** sobre la referencia al archivo **ejes.js**; aceptamos crear dicho archivo. Agregamos el código base, que detectará cuando la aplicación web ha finalizado su carga en el navegador:

Dentro del **Event Listener** declarado, añadimos la primera línea de código:

```
document.addEventListener("DOMContentLoaded", function() {  
    ...  
})
```

```
var imagen = document.querySelector("img");
```

De esta forma, a través de la variable **imagen**, mediante **querySelector** capturamos el objeto HTML correspondiente a la imagen que queremos rotar, según el movimiento que detectará el acelerómetro del dispositivo. Después, declaramos otro **Event Listener**, que escuchará cuando se produzca el movimiento del dispositivo móvil, utilizando **deviceorientation**. Al detectar dicho movimiento, mediante el parámetro **e**, parametrizará los valores en tiempo real que el acelerómetro genere:

```
window.addEventListener('deviceorientation', function(e) {
    ...
});
```

Utilizamos las variables **a**, **b** y **g** para almacenar los valores de Alpha, Beta y Gamma, respectivamente. Dado que los números devueltos a través de estos parámetros pueden ser del tipo **double**, utilizaremos el método **Math.floor()** para redondearlos hacia su entero inferior más próximo.

```
a = Math.floor(e.alpha);
b = Math.floor(e.beta);
g = Math.floor(e.gamma);
```

El parámetro **e** se transforma en un objeto, el cual almacena las propiedades **alpha**, **beta** y **gamma**, junto con sus valores en tiempo real. Esto es lo que debemos leer a continuación y almacenar dichos valores en diferentes variables:

Ahora un poco de CSS

Para terminar de dar formato a nuestro script, invocaremos el método CSS **rotate()** relacionado a las transformaciones que incluye CSS3, para mover el producto acorde al valor que recibimos de cada eje.

Los métodos que utilizaremos son **rotateZ()**, **rotateX()** y **rotateY()**, y pasaremos el valor de los respectivos enteros almacenados en las variables **a**, **b** y **g**, seguido de la leyenda **deg** (**degrees**, o **grados** en inglés):

```
var transform = 'rotateZ(' + Math.round(e.alpha) + 'deg) rotateX(' + Math.
round(e.beta -90) + 'deg) rotateY(' + Math.round(e.gamma) + 'deg');
```

En la variable **transform** agrupamos la invocación a estos métodos CSS, concatenando los valores que deseamos aplicar, a cada uno de los métodos CSS. Una vez almacenado todo esto en la variable mencionada, le aplicamos la transformación CSS a la imagen que deseamos rotar:

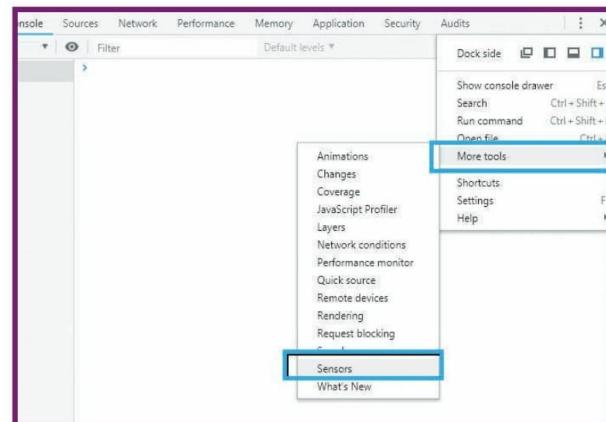
```
imagen.style.transform = transform;
```

Con este código, ya podemos probar nuestro nuevo proyecto. La estructura completa de cómo debe quedar el código puede verse en la imagen.

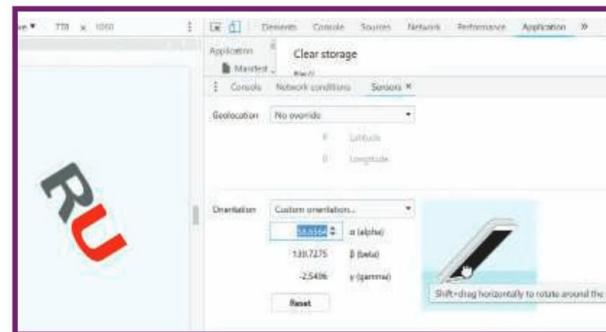
```
archivo Editar Selección Ver Ir Depurar Terminal Ayuda ejes.js - Ejes - Visual Studio Co...
JS ejes.js x
1 document.addEventListener("DOMContentLoaded", function() {
2     var imagen = document.querySelector("img");
3
4     window.addEventListener('deviceorientation', function(e) {
5         a = Math.floor(e.alpha);
6         b = Math.floor(e.beta);
7         g = Math.floor(e.gamma);
8         var transform = 'rotateZ(' + Math.round(e.alpha) + 'deg) rotateX(' + Math.round(e.beta -90) + 'deg) rotateY(' + Math.round(e.gamma) + 'deg)';
9         imagen.style.transform = transform;
10    });
11 })
```

Probar la rotación de la imagen

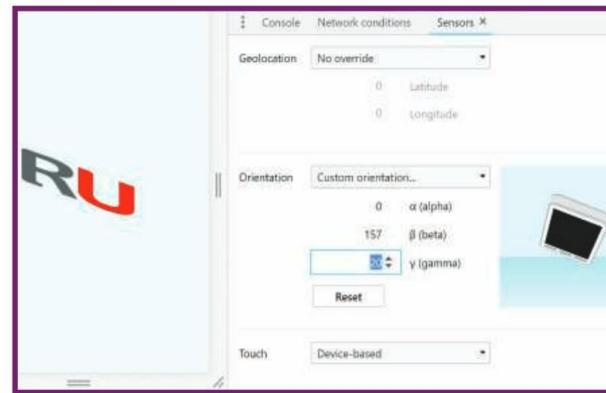
1 Si contamos con un servidor web en nuestro equipo, lo iniciamos y, con la dirección IP y el puerto, cargamos dicha ruta en nuestro dispositivo móvil para comenzar las pruebas. Si no disponemos de un servidor web, podemos optar por las **Herramientas para el desarrollador** de Google Chrome, que nos permitirá probar localmente este proyecto emulando la rotación de un dispositivo. En este último caso, cargamos el proyecto en Google Chrome y presionamos la tecla F12, que inicia las herramientas mencionadas.



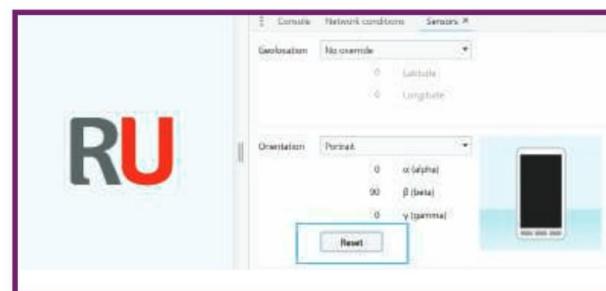
2 En el apartado **Sensors** de las Herramientas para el desarrollador de Google Chrome, encontramos la opción **Orientation**. Por defecto, esta viene en modo **OFF**, o sea, apagado. Desplegamos el Combo desplegable y seleccionamos **Custom Orientation**. A continuación, se habilitará el PAD que contiene la imagen de un teléfono móvil, junto con los campos relacionados a los ángulos **Alpha**, **Beta** y **Gamma**.



3 Para comenzar a rotar la imagen, debemos pulsar con el mouse sobre el PAD mencionado y, sin soltar el botón primario, comenzamos a arrastrar el gráfico del teléfono móvil hacia alguno de sus lados o en forma diagonal (modo Drag and Drop). Esto hará que la imagen contenida en el documento HTML empiece a girar. Si anulamos el efecto de arrastre, la imagen quedará fija en la posición donde nos detuvimos.



4 Otra opción manual para rotar la imagen es seleccionar alguno de los campos de texto disponibles e ingresar un valor numérico que puede ir de 0° a 180°, o de -0° a -180°. De esta forma, veremos que la imagen rota de forma fija. Podemos ir alterando los diferentes campos **Alpha**, **Beta** y **Gamma** con números dentro de los parámetros indicados, y así apreciar la rotación del gráfico.



5 Luego de realizar todas las pruebas que creamos pertinentes, podemos volver la imagen a 0 (cero), restableciendo así su valor inicial. Para hacerlo, basta con pulsar el botón **RESET**. Entonces podremos volver a realizar pruebas y correcciones, o alternar entre las pruebas descriptas en los pasos 02 y 03.

Manejo de datos remotos con JSon



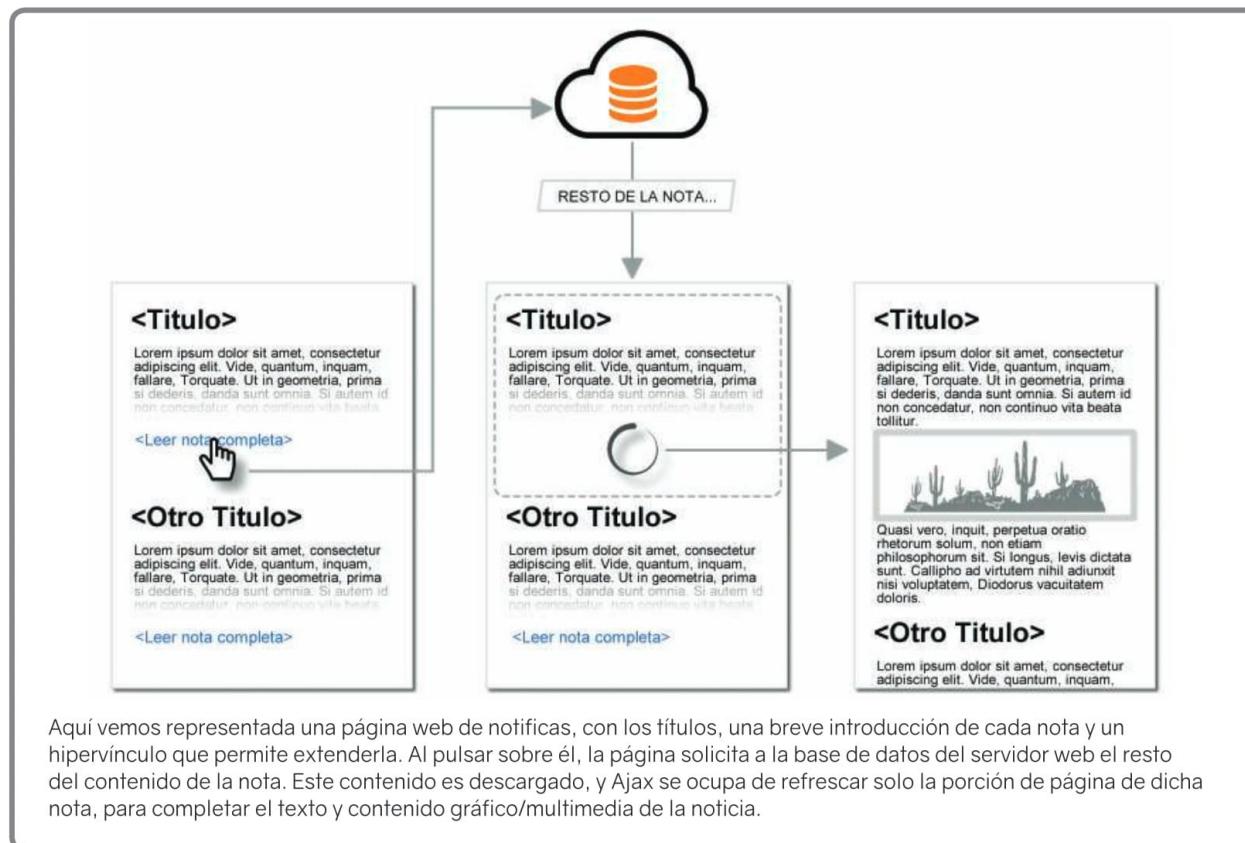
A continuación nos introduciremos en el mundo del acceso a datos remotos. Veremos cómo aprovechar la técnica Ajax para acceder a datos del tiempo, leyendo los resultados en formato JSon y visualizándolos de manera clara para el usuario.

Desde la llegada de Internet, las técnicas de acceso a datos remotos a través de sitios web y aplicaciones móviles se ha tornado diferente respecto al que los desarrolladores de software estaban acostumbrados a ver.

A través de este nuevo proyecto, accederemos a la información remota de un servicio web de pronóstico del tiempo y aprenderemos de qué forma podemos sacar provecho de la tecnología Ajax.

Ajax

La técnica Ajax es un acrónimo de **Asynchronous JavaScript And XML** para el desarrollo web, que nos permite crear aplicaciones interactivas. Estas se ejecutan del lado cliente (usuarios), mientras que los datos (remotos) son accedidos a través de una comunicación asíncrona contra el servidor que los aloja, en segundo plano. Esta técnica fue desarrollada originalmente por **Microsoft** bajo su concepto de **scripting remoto**, en 1995, y se introdujo en el mercado en 1998. El nombre Ajax llegó casi una década después de la mano de Jesse James Garrett.



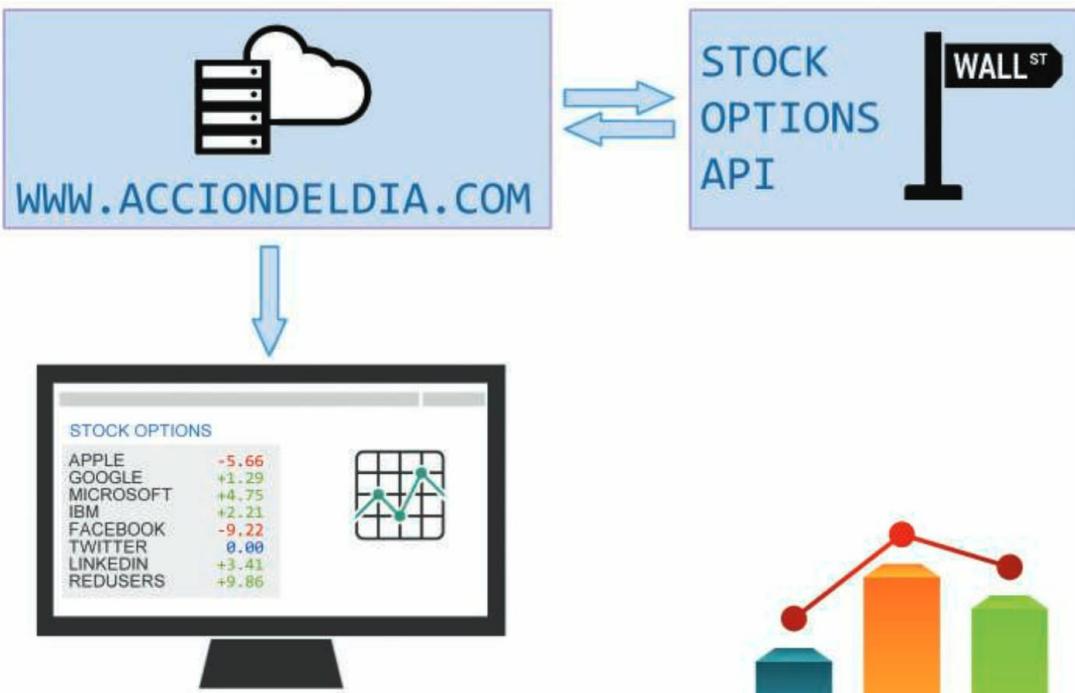
Beneficios

Esta técnica es útil, por ejemplo, para cargar rápidamente un sitio web volcando información parcial a él; luego, Ajax descarga el resto de manera asíncrona. De este modo, no solo la web carga con rapidez, sino que también se descargan porciones de información desde el servidor, evitando que éste se sature. Y, por último, no se refresca todo el documento web, sino únicamente la porción donde se agrega el contenido. Así logramos también que el usuario no consuma datos excesivos ni espere demasiado tiempo. Este tipo de intercambio de información se popularizó para bien, con el objetivo de reducir el envío de datos y evitar la saturación de servidores cuando las consultas por parte de los usuarios son masivas.

Web services

La técnica de acceso a datos remotos se realiza a través de lo que se conoce como web services. Ajax solo permite acceder a datos de forma sincrónica, es decir, a aquellos que se encuentren alojados en el mismo

servidor de donde proviene la aplicación web. Únicamente se puede acceder a servidores externos o de terceros si estos soportan la tecnología **CORS** (acrónimo de **Cross Origin Resource Sharing**).



En el diagrama podemos ver la representación del acceso mediante Ajax a la información de acciones en tiempo real. Como usuarios, accedemos a un sitio web específico; este, a su vez, consulta la API de Wall Street para obtener la cotización de acciones del día, actualizadas al instante recurriendo a ellas mediante la tecnología CORS.

Devolución de datos

Por lo general, las APIs que consultamos mediante Ajax devuelven los datos en un formato determinado. Entre aquellos que podemos elegir, contamos con los dos más populares: **XML** (acrónimo de eXtensive Markup Language) y **JSon** (JavaScript Object Notation). Si bien son los formatos más populares en el mundo del acceso a datos remotos, JSon ha sido el más impulsado este último tiempo, por ser mucho más liviano en estructura y contenido, que lo que originalmente es un archivo del tipo XML.



JSon es usado también para construir menús de navegación web. En lugar de modificar un HTML o leer una BBDD se modifica un archivo JSon, y, al recargar la página, veremos la opción de menú.

La aplicación del clima

Para entender técnicamente el acceso y lectura de datos remotos mediante JSon, desarrollaremos un ejercicio práctico que nos permita obtener la información del clima de un servidor remoto. Para esto, tendremos por un lado la aplicación web, que tiene su frontend ya desarrollado, y donde solo deberemos ocuparnos de escribir el código que nos conecte a los datos remotos.

OBJETIVO DE LA APLICACIÓN

A través del siguiente diagrama, veamos qué puntos de la aplicación HTML utilizaremos para mostrar los datos en formato JSon, obtenidos a través de Ajax.



Almacenamiento de datos, construcción de menús, compartir noticias, y referencia a contenidos específico para web o móviles, son opciones que se implementan a través de JSon.

OpenWeather Map

Para visualizar los datos del clima, utilizaremos el servicio gratuito de la API de **Open Weather Map**. Veamos a continuación los pasos para conseguirlo:

Create New Account

Fernando O.

fernando@vidamobile.com.ar

.....

.....

We will use information you provided for management and administration purposes, and for keeping you informed by mail, telephone, email and SMS of other products and services from us and our partners. You can proactively manage your preferences or opt-out of communications with us at any time using Privacy Centre. You have the right to access your data held by us or to request your data to be deleted. For full details please see the OpenWeather [Privacy Policy](#).

I am 16 years old and over

I agree with [Privacy Policy](#), [Terms and conditions of sale](#) and [Websites terms and conditions of use](#).

I consent to receive communications from OpenWeather Group of Companies and their partners:

1 Ingresamos en la web oficial, www.openweathermap.org, y ubicamos en el extremo superior de la página el apartado **SIGN UP**. Esto nos llevará al formulario principal que debemos completar para conseguir una suscripción al servicio de Open Weather Map. Brindamos los datos solicitados para poder acceder a la API del servicio.

Weather in your city Hello Fernando O.

How and where will you use our API?

Hi! We are doing some housekeeping around thousands of our customers. Your impact will be much appreciated. All you need to do is to choose in which exact area you use our services.

Company: Vidamobile

* Purpose: Education/Science

Cancel Save

Try our simple and fast APIs to satellite imagery, weather data and more.

- Satellite imagery archive (True & False color, NDVI & EVI indices)
- Weather (current data, forecast and history)
- Accumulated temperature and precipitation

2 Es posible que el sistema nos solicite validar nuestro e-mail. Consultemos la casilla de correo de la cuenta utilizada para ver si nos sugiere realizar una validación. Finalizado este paso, es factible que se nos pidan datos adicionales, como el nombre de nuestra empresa o proyecto, y el segmento al cual apuntamos. En nuestro ejemplo, seleccionamos como propósito el ítem **Educación/Ciencia**.





OpenWeatherMap

Weather Maps API Price Partners Stations Widgets News About

New Products Setup API keys Services Payments Billing plans Block logs History bulk Logout

You can generate as many API keys as needed for your subscription. We accumulate the total load from all of them.

Key	Name	Create key
524	Default	<input type="button" value="Create key"/> <input type="text" value="libro_javascript"/> <input type="button" value="Generate"/>

3 Accedemos a la plataforma. Ubicamos el apartado **API-Keys**, a través del cual podemos crear una o más claves, que serán requeridas al momento de invocar la API del clima. Esta clave o **Key** que nos solicita nos dará una idea de cómo se comporta el resto de los servicios JSON que podemos acceder de la misma forma. Ingresamos para dicha clave un nombre de proyecto lo más descriptivo posible.

OpenWeatherMap

Weather Maps API Price Partners Stations Widgets News About

New Products Setup API keys Services Payments Billing plans Block logs History bulk Logout

Name	Description	Price plan	Limits	Details
Weather	Current weather and forecast	Free plan	Threshold: 7,200 Hourly forecast: 5 Daily forecast: 0 Calls 1min: 60	view

4 Finalmente, veremos una o más claves creadas en una grilla de datos bastante intuitiva, dentro del apartado **Services**. La columna **Price plan** indica el tipo de plan que tenemos (hay gratuitos y pagos), y en la columna **Limits** podremos acceder a un registro rápido de la cantidad de solicitudes que hemos invocado a la API, y la cantidad máxima que tenemos disponible para consumir.

Continuemos ahora con la declaración de las variables correspondientes para capturar los elementos HTML necesarios y así poder visualizar en ellos los datos del tiempo obtenidos.

Crear una web para el pronóstico del tiempo

Con todos los pasos de suscripción completos, comenzemos a darle vida a nuestro proyecto. Para hacerlo, descargamos el archivo base **Proyecto-climAtica-Base.zip**, alojado en el repositorio de esta obra. Lo descomprimimos en una carpeta y creamos un nuevo proyecto en Visual Studio Code.



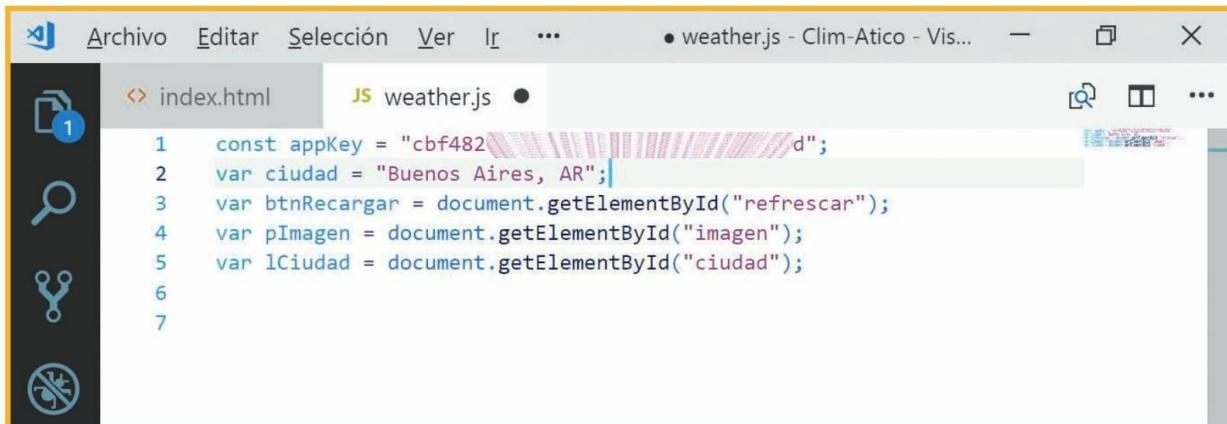
Al editar el documento HTML, encontraremos en el apartado **<head>** la referencia al archivo **weather.js**. Hacemos CTRL + clic sobre él y seleccionamos **Crearlo**. A continuación, escribimos el siguiente código:

```
const appKey = "INGRESA_TU_API_KEY_AQUI";
var ciudad = "Buenos Aires, AR";
var btnRecargar = document.
getElementById("refrescar");
var pImagen = document.getElementById("imagen");
var lCiudad = document.getElementById("ciudad");
```

Lo primero que hacemos es declarar la variable **appKey**, la cual será enviada como parámetro a la URL invocada para obtener los datos del clima. En este caso, usamos **const** en vez de **var**, para que esta no pueda ser modificada en ningún momento. Luego declaramos la variable **ciudad**, donde almacenamos el nombre de la ciudad que buscamos. Es conveniente incluir el código

internacional del país (usualmente coincide con la terminación de un dominio web: AR, US, CL, CN, etc.). Esto evitara que recibamos el clima de una ciudad con el mismo nombre que la buscada, pero de otro país.

Capturamos los elementos **refrescar**, **imagen** y **cargando** en las variables **btnRecargar**, **pImagen** y **progreso**, respectivamente.



Continuemos ahora con la declaración de las variables correspondientes para capturar los elementos HTML necesarios y así poder visualizar en ellos los datos del tiempo obtenidos:

```
var lTemp = document.getElementById("temperatura");
var lHume = document.getElementById("humedad");
var lPres = document.getElementById("presion");
var lVien = document.getElementById("viento");
var lPron = document.getElementById("descripcion");
var lMima = document.getElementById("minimax");
var lUact = document.getElementById("actualizacion");
var Rueda = document.getElementById("rueda");
```

A través de este bloque de código, declaramos cada una de las variables necesarias donde, a posteriori, le asignaremos el valor correspondiente a los datos del clima obtenidos. Declaremos ahora la función que controlará el elemento Preloader:

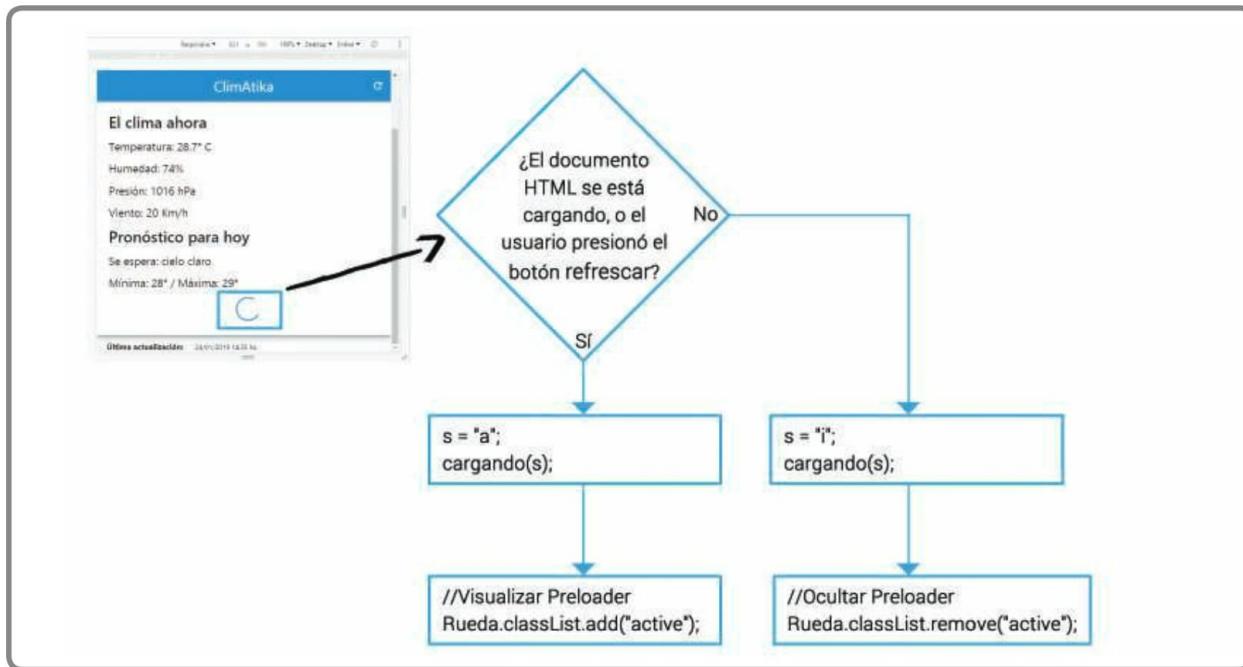
```
function cargando(s) {
switch(s) {
case "a": Rueda.classList.add("active"); break;
case "i": Rueda.classList.remove("active"); break;
}
}
```

Esta función es la que se ocupa de mostrar u ocultar el componente HTML **Preloader**.

Recibe un parámetro (**s** = status), que le indicará cuándo debe activar el componente (**s = "a"**), o cuándo ponerlo inactivo (**s = "i"**). Para realizar esto, mediante la variable Rueda, capturamos el ID del elemento HTML **rueda** y le agregamos la clase **Materialize CSS active**, utilizando el método **add()** de la propiedad **classList**.

Para dejar de mostrar el elemento Preloader, simplemente quitamos la clase **active** mediante el método **remove()** de la propiedad anteriormente mencionada.





```
btnRecargar.addEventListener("click", obtenerDatosDelClima);
```

Este Event Listener se ocupará de ejecutar la función **obtenerDatosDelClima()** cuando el usuario haga clic en el botón **refrescar** (previamente atado a la variable **recargar**). Este mismo evento deberá ser llamado cuando el documento HTML finalice su carga. Para esto, creamos un **addEventListener** relacionado:

```
document.addEventListener("DOMContentLoaded", obtenerDatosDelClima);
```

Esta función se ocupará de obtener los datos remotos del clima, y será invocada también cuando finalice la carga del DOM HTML. Vamos entonces a crearla:

```

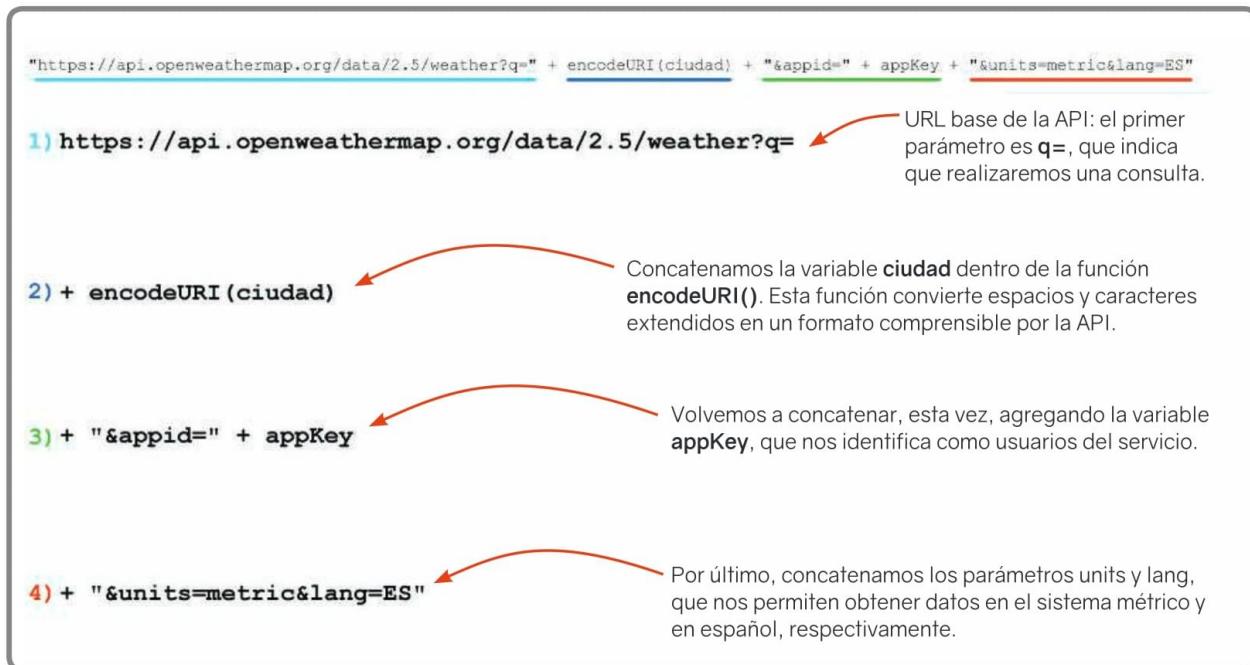
function obtenerDatosDelClima() {
    cargando("a");
    if (ciudad.value === "") {
        alert("Debe ingresar una búsqueda valida.");
    } else {
        var URLowm = "https://api.openweathermap.org/data/2.5/weather?q=" +
        encodeURI(ciudad) + "&appid=" + appKey + "&units=metric&lang=ES";
        peticionHTTPAsincronica(URLowm, respuesta);
    }
}

```

Lo primero que hacemos es setear el atributo del componente HTML **progreso**, en modo “visible”. Esto permitirá ver un componente **preloader** mientras se ejecuta la acción de obtención de datos.

URL del web service a invocar

Validamos que la variable ciudad contenga datos asociados. De ser correcto, creamos la variable **URLowm** (del acrónimo URL Open Weather Map), donde almacenamos la URL que invoca la API para obtener los datos remotos en formato JSoN. Analicemos en el siguiente diagrama la composición de dicha URL:



Ya contamos con la función **obtenerDatosDelClima()**, que inicia el componente Preloader, valida que la variable ciudad contenga datos, y arma la URL del servicio web de Open Weather Map para obtener los datos del clima de la ciudad. A continuación, creamos la función **peticionHTTPAsincronica()**, que será invocada para iniciar la obtención de datos.

El objeto XMLHttpRequest()

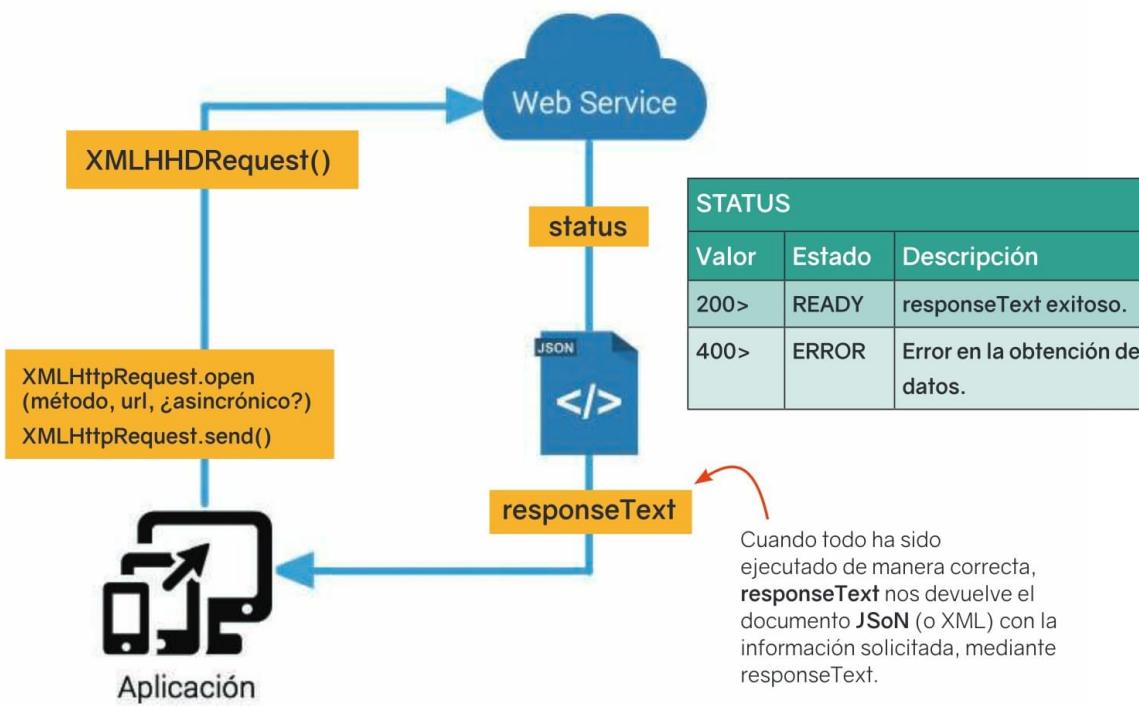
```
function peticionHTTPAsincronica(url, callback) {
  var req = new XMLHttpRequest();
  req.onreadystatechange = () => { if (req.readyState == 4 && req.status == 200)
    callback(req.responseText);
  req.open("GET", url, true); //método asincrónico
  req.send();
}
```

En la función anterior, tenemos una variable **req** que inicia el objeto **XMLHttpRequest()**. Este objeto proporciona la manera más fácil para obtener información remota (de una URL o web service), sin tener que recargar toda la página para mostrarla (Ajax). Con los datos obtenidos, nos ocuparemos de actualizar parte

del documento HTML, sin interrumpir lo que el usuario está haciendo o visualizando. Este objeto cuenta con una serie de propiedades, de las cuales por ahora nos interesa utilizar **readyState**, **onreadystatechange**, **status** y **responseText**. Veamos a continuación la representación gráfica de esta función.

READYSTATE

Valor	Estado	Descripción
0	UNINITIALIZED	Todavía no se llamó a open().
1	LOADING	Todavía no se llamó a send().
2	LOADED	send() fue invocado. Encabezados y Estado se encuentran disponibles.
3	INTERACTIVE	Descargando; responseText contiene información parcial.
4	COMPLETED	La operación está terminada.



PARÁMETROS DE OPEN(...)

Valor	Descripción
Método	GET o PUT
URL	Correspondiente al web service a consultar.
ASYNC	TRUE para indicar que la consulta se ejecutará de forma asincrónica.

La masificación de las aplicaciones web, web mobile y mobile nativas, hace que la mayoría de las consultas de datos remotos basen los estados de la solicitud (espera, recepción y visualización), en el modelo asincrónico.

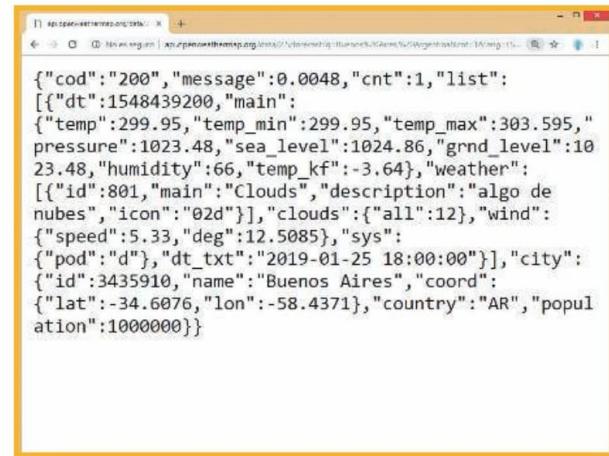


Formateo de datos JSon

Finalmente, con el documento en formato JSon que recibimos mediante **responseText**, nos queda leer los datos contenidos en él para mostrarlos en pantalla. Antes de escribir el siguiente código, entendamos un poco más el formato de datos JSon con el cual trabajaremos.

Cuando el código antes visto se ejecuta correctamente, nos devuelve un documento JSon, que contiene los datos con los cuales trabajaremos. El formato de datos enviado es similar al que visualizamos en la figura.

Si lo leemos algunas veces, analizando los datos contenidos, el set de información devuelta nos parecerá fácil de comprender. Pero este en sí no es el mayor problema. El verdadero inconveniente puede darse cuando queremos utilizar el método **parse()** para trasladar dichos datos a una variable que luego utilizaremos para tomar el dato y mostrarlo en pantalla. Nos sucederá esto no solo con el formato JSon devuelto por Open Weather

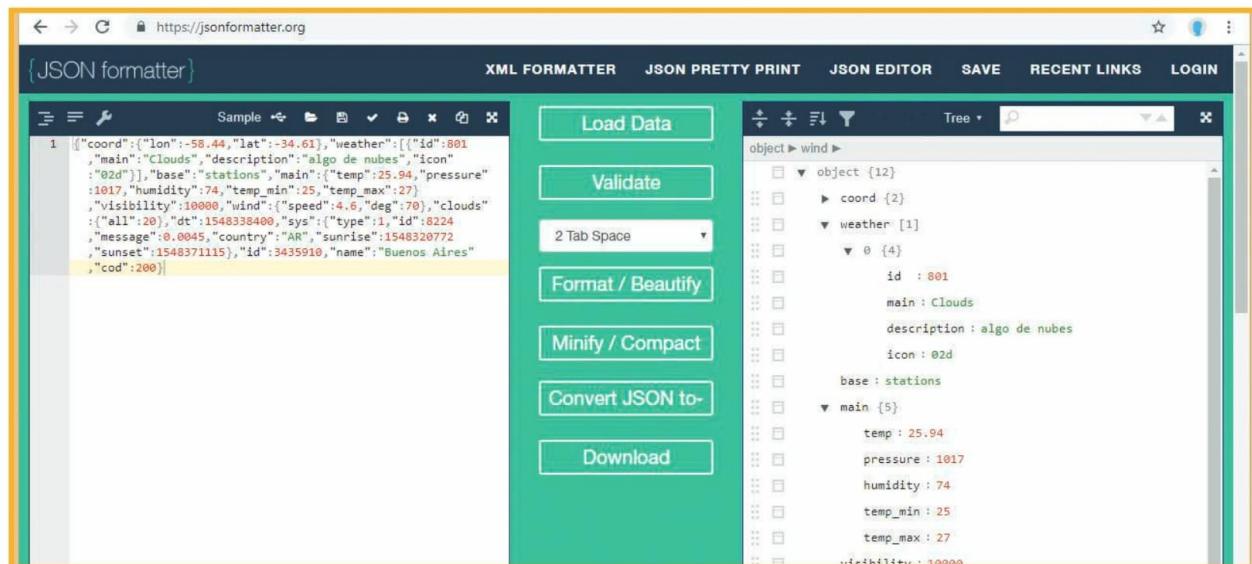


```
{"cod": "200", "message": 0.0048, "cnt": 1, "list": [{"dt": 1548439200, "main": {"temp": 299.95, "temp_min": 299.95, "temp_max": 303.595, "pressure": 1023.48, "sea_level": 1024.86, "grnd_level": 1023.48}, "humidity": 66, "temp_kf": -3.64}, "weather": [{"id": 801, "main": "Clouds", "description": "algo de nubes", "icon": "02d"}], "clouds": {"all": 12}, "wind": {"speed": 5.33, "deg": 12.5085}, "sys": {"pod": "d"}, "dt_txt": "2019-01-25 18:00:00"}, "city": {"id": 3435910, "name": "Buenos Aires", "coord": {"lat": -34.6076, "lon": -58.4371}, "country": "AR", "population": 1000000}]} 
```

Map, sino también con cualquier otro servicio que devuelva datos en este formato o en **XML**. Por eso, para hacer mucho más fácil su lectura y aplicación mediante **parse()**, recomendamos copiar todos estos datos y formatearlos con un intérprete de documentos JSon.

JSon Formatter

Existen muchos programas que pueden ayudarnos a realizar este cometido. Uno de ellos es accesible desde la URL [www.jsonformatter.org](https://jsonformatter.org). Ingresando en ella, en el lateral izquierdo pegamos el contenido devuelto por Open Weather Map. Pulsamos luego la opción **Format / Beautify**, y así obtendremos en la columna derecha una estructura de árbol de los datos JSon para nuestra aplicación. Esta estructura está anidada, y nos permitirá navegarla fácilmente para comprender de dónde debemos tomar cada dato que visualizaremos en nuestra aplicación HTML del clima.



The screenshot shows the JSON formatter interface. On the left, there is a code editor containing a JSON object. On the right, there is a tree view showing the hierarchical structure of the JSON data. The tree view includes nodes for 'wind', 'coord', 'weather', 'base', 'main', and various numerical values for temperature, pressure, humidity, and visibility.

```
object > wind >
  object > coord {12}
    object > main {2}
      object > weather [1]
        object > 0 {4}
          id : 801
          main : Clouds
          description : algo de nubes
          icon : 02d
          base : stations
          main : 25.94
          pressure : 1017
          humidity : 74
          temp_min : 25
          temp_max : 27
          visibility : 10000
        
```

Lectura de datos JSoN recibidos

Hasta el momento tenemos resuelta la declaración de variables de nuestra aplicación, la captura de eventos de actualización o carga automática de datos, y el armado de contenido y petición de datos al servidor. Ahora, con la recepción correcta de dichos datos, nos queda

leer el contenido y mostrarlo en pantalla. Para lograrlo, vamos a crear una función denominada **respuesta()**, la cual recibirá un parámetro (**r**) que será, por supuesto, el documento JSoN recibido del web service del pronóstico del clima:

```
function respuesta(r) {
  ...
}
```

Pero ¿cómo recibimos aquí dicha respuesta? Si miramos la función **obtenerDatosDelClima()**, veremos que dentro de su código invocamos a la función **peticionHTTPAsincronica()**, a la cual le indicamos un parámetro **URLowm** con la URL del servicio web a invocar, y un segundo parámetro **respuesta** (es la función que estamos creando ahora), que en realidad es una función de retorno (devuelta por **callback** por esta última función). Los datos que retornan son los datos en formato JSoN. Declaremos a continuación, dentro de la función **respuesta()**, la variable **clima**, utilizando **let** en lugar de **var**:

```
let clima = JSON.parse(r);
```

A **clima** le asignamos el objeto **JSON**, seguido del método **parse()**. Este método recibe el objeto JSON devuelto por el servicio web, y lo transforma en un objeto JavaScript, para que podamos leerlo de manera más amigable. Sigamos con el código:

```
lCiudad.innerHTML = clima.name + ", " + clima.sys.country;
```

Aquí comenzamos con la asignación de los datos correspondientes al objeto JSON, a los componentes HTML de nuestra aplicación. En primera instancia, asignamos el nombre de la ciudad recuperada y del país correspondiente, para mostrarlo como título principal en la aplicación HTML (esto nos asegura que la ciudad sea la que realmente queremos consultar). Ahora agregamos los datos del clima en sí:

```
lTemp.innerHTML = "Temperatura: " + clima.main.temp.toFixed(1) + "° C";
lHume.innerHTML = "Humedad: " + parseInt(clima.main.humidity) + "%";
lPres.innerHTML = "Presión: " + parseInt(clima.main.pressure) + " hPa";
```

Ya tenemos en pantalla la información de ciudad, país, temperatura, humedad y presión atmosférica. Como invocamos al servicio web pasándole el parámetro **units=metric**, recibimos los datos de la temperatura en **grados centígrados**, y la presión en **hectopascales**. Continuemos con el código:

```
lVien.innerHTML = "Viento: " + parseInt(clima.wind.speed * 3.6) + " Km/h";
```

Los datos del viento los recibimos en **metros/segundos**. Para convertirlos a **Km/h**, simplemente multiplicamos el valor recibido por **3.6**. Utilizamos la función **parseInt()** para evitar en algunos resultados la visualización de valores en decimales:



```
lPron.innerHTML = "Se espera: " + clima.weather[0].description;  
lMima.innerHTML = "Mínima: " + parseInt(clima.main.  
temp_min) + "° / Máxima: " + parseInt(clima.main.temp_max) + "°";
```

Finalmente, mostramos una breve descripción del clima que se espera, y las temperaturas mínimas y máximas del día. Ahora nos ocuparemos de visualizar una imagen acorde al clima que estamos presentando. Open Weather Map nos devuelve en el parámetro **imagen** un dato en código, el cual corresponde al nombre de la imagen en sí, alojada en sus servidores. Lo único que debemos hacer es armar la URL hacia la ruta correcta, pasando en ella el código recibido por JSON:

```
pImagen.src = "http://openweathermap.org/img/w/" + clima.weather[0].icon + ".png";  
pImagen.setAttribute("width", "90px");
```

Con **setAttribute()** agrandamos un poco el tamaño de la imagen original, que es de 32 píxeles, llevándola a 90px. Esta es una dimensión adecuada para verse mejor, sin llegar a deformar o pixelar la imagen en sí.

```
pImagen.src = "http://openweathermap.org/img/w/  
+ clima.weather[0].icon + ".png";
```



Concatenamos el **código de ícono** recibido mediante JSON, con la **URL** de repositorio de íconos de OWN, para así obtener la imagen que representa el clima actual obtenido

```
pImagen.setAttribute("width", "90px");
```



Luego, seteamos el atributo "**ancho**" de la imagen a **90 px**, para poder mejorar su visualización, sin pixelarla.

Por último, agregamos el siguiente código:

```
lUact.innerHTML = ultimaActualizacion();
cargando("i");
```

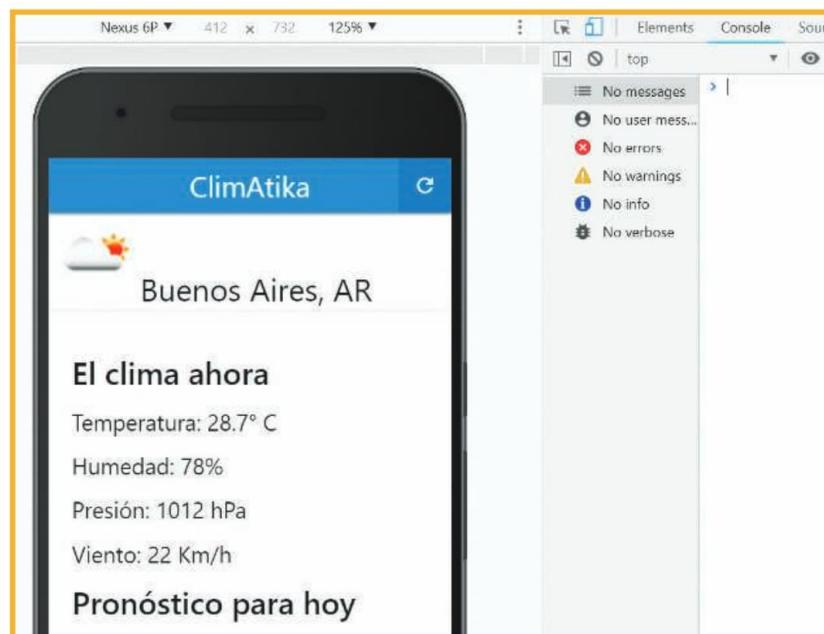
Lo que hacemos aquí, en la primera línea, es mostrar la fecha y hora de esta actualización del clima recibida. Esto se hace a través de la función **ultimaActualizacion()**, que crearemos a continuación. Por último, desactivamos el componente HTML Preloader, con esta función creada para tal fin:

```
function ultimaActualizacion() {
d = new Date();
f = d.getDate(); if (f < 10) {f = "0" + f};
m = (d.getMonth() + 1); if (m < 10) {m = "0" + m};
actualizacionFecha = f + "/" + m + "/" + d.getFullYear();
actualizacionHora = d.getHours() + ":" + d.getMinutes() + " hs.";
return actualizacionFecha + " " + actualizacionHora;
}
```

Esta función solo crea un objeto **Date()**, recupera los datos de la fecha y hora del momento, y devuelve las variables formateadas, para que los datos consultados se muestren en pantalla. Con esto, ya completamos nuestro proyecto JS. Vamos a probarlo para validar que funciona y que conseguimos obtener los datos del clima de nuestro proveedor Open Weather Map.

La prueba de fuego

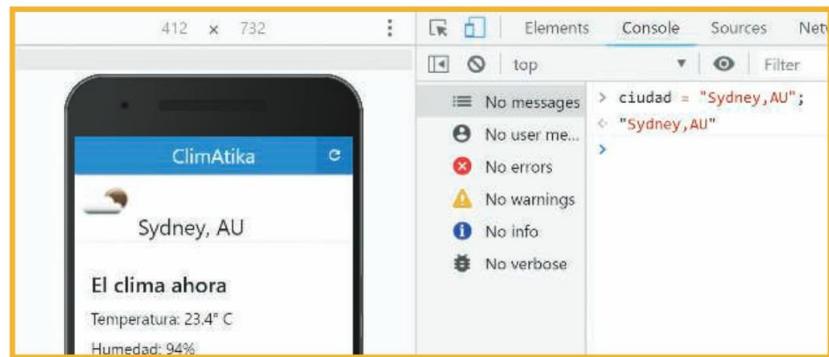
Una vez que finalizamos nuestro aporte JS para darle vida a la obtención de datos remotos, solo nos resta probar el funcionamiento correcto del proyecto. Lo ejecutamos en Google Chrome y vemos si se comporta como corresponde, cargando en el inicio los datos del clima.



Si no lo hicimos aún, presionemos la tecla F12 para iniciar las Herramientas del desarrollador. A través de ellas podremos cambiar manualmente la ciudad que deseamos consultar; solo escribimos en la pestaña Console, lo siguiente:

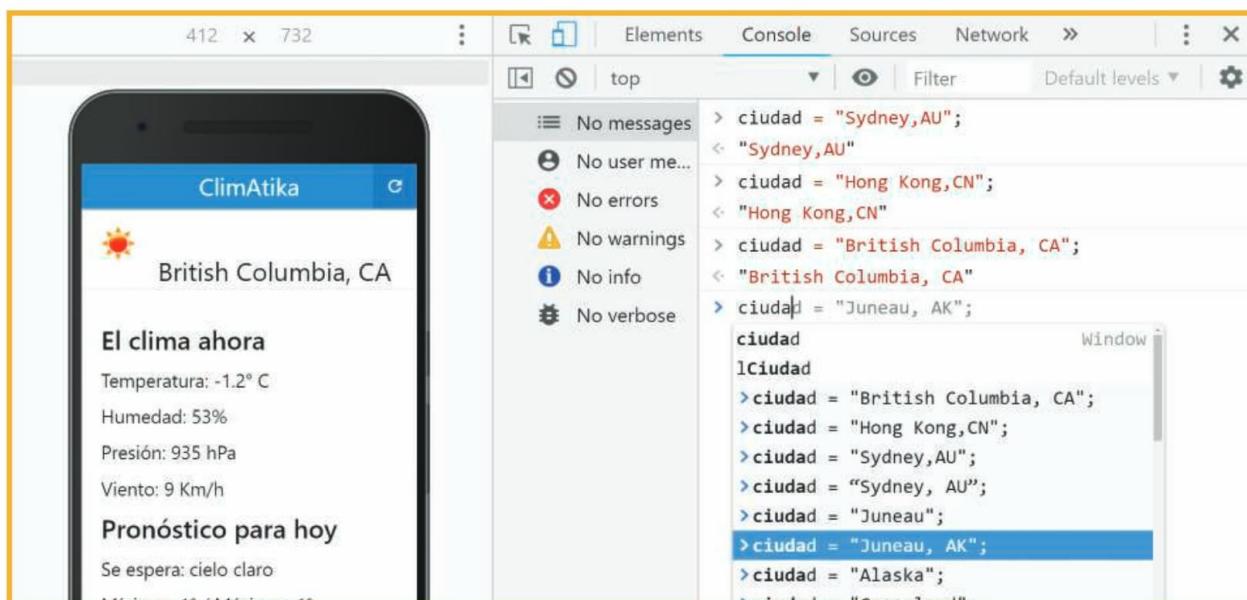
```
ciudad = "Sydney, AU";
```

Luego, pulsamos ENTER y refrescamos los datos presionando el botón superior derecho, para ver que llegue la información del país ingresado.



Podemos repetir este último paso intercambiando entre diferentes ciudades. Pero tengamos cuidado, porque, en algunos casos, Open Weather Map es sensible y, como bien dijimos en su momento, podemos encontrar que ingresamos los datos de

una ciudad y nos devuelve otra. Un ejemplo de esto sería escribir "**Singapur**". El servicio traerá los datos de esta ciudad, ubicada en **México**. Para obtener información de la ciudad malaya, debemos ingresar "**Singapore**".



Mejorar el desarrollo

Ahora, para poder consultar el clima de otros sitios, sin tener que cambiar por Consola la ciudad de nuestro interés, ¿se animan a integrar un cuadro de diálogo que pregunte y guarde cada ciudad ingresada? Como tip, recomendamos implementar almacenamiento local.

Proteger el código mediante ofuscación



En este último proyecto, repasaremos las diferentes herramientas online e instalables que nos permiten generar una capa de protección en nuestro código JS, sabiendo que en la WWW este estará expuesto fácilmente al acceso público.

Para entender bien el significado de la ofuscación, veamos los siguientes bloques de código.

CÓDIGO A

```
// Crear una función para saludar
function saludo() {
  console.log("Hola, mundo");
}
saludo();
```

¿Notamos la diferencia entre un código y el otro, verdad?

¿Llegaremos al mismo resultado si ejecutamos dentro de un proyecto el **Código A** y luego el **Código B**?

La respuesta para esto último es: sí, conseguiremos el mismo resultado en el caso de ejecutar estos códigos por separado dentro de un mismo proyecto.

CÓDIGO B

```
var
_0x587e=['Hola,\x20mundo'];(function(_0x54e271,_0x19c
d06){var
_0x3844e4=function(_0x1d2287){while(--_0x1d2287){_0x
54e271['push'](_0x54e271['shift']());}};_0x3844e4(_+_
0x19cd06);}(_0x587e,0x1d6));var
_0x13a0=function(_0x7b168c,_0x434660){_0x7b168c=_0x7b
168c-0x0;var _0x246fa4=_0x587e[_0x7b168c];return
_0x246fa4;};function
saludo(){console['log'](_0x13a0('0x0'));}saludo();
```

Ofuscación

La definición de ofuscación relacionada al ámbito informático se puede traducir como "el acto deliberado de realizar un cambio no destructivo en el código fuente de un software o en el código máquina, cuando el programa se encuentra en modo binario o compilado, por el mero fin de que no sea fácil de entender o leer" (extracto del artículo de Wikipedia: <https://es.wikipedia.org/wiki/Ofuscación>).

¿Pero cuál es el propósito? El propósito principal del comportamiento de un sistema de ofuscación de código es proteger el código fuente de una aplicación.

Si bien JavaScript suele usarse para manejar el comportamiento del frontend de una web, en determinados casos maneja datos más sensibles: **Token**, **AppKey**, **UIDs**, acceso a **APIs**, etcétera.



Herramientas online

Existe un sinfín de herramientas en línea que nos permiten llevar a cabo el proceso de ofuscación. Este no está limitado solo a JavaScript; también puede aplicarse en determinados casos a CSS y HTML, aunque no es tan necesario para ellos.

Una de las herramientas en línea para llevar a cabo nuestro propósito de ofuscar código es Obfuscator.io, disponible en <https://obfuscator.io>. Es muy fácil de utilizar.

Para entenderla bien, descarguemos del repositorio de archivo el proyecto **Ejemplo-Ofuscacion.zip**. Lo descomprimamos y abrimos en Visual Studio Code.

Al ejecutarlo, veremos que solo tiene un botón, que cuando lo presionamos nos devuelve un mensaje con el clásico “**Hola mundo!**”.

Vamos a editar el archivo JS contenido en este proyecto, copiar todo el código e ingresar en

The screenshot shows a browser window with the title "Ofuscación de código". Below the title, it says "Prueba de ofuscación". There is a button labeled "Saludar" and a message area containing the text "Esta página dice" followed by "Hola mundo!".

el link de la aplicación online de ofuscación. Nos desplazamos hasta el apartado **Copy and Paste JavaScript Code** y pegamos el código fuente de nuestro proyecto en dicha ventana (reemplazando el que está de muestra). Pulsamos a continuación el botón **Obfuscate**, y vemos el resultado de nuestro código JS, ya procesado y ofuscado.

The screenshot shows the "JavaScript Obfuscator Tool" interface. In the "Copy & Paste JavaScript Code" section, there is an obfuscated JavaScript code block. Below it, there are several configuration options: "Compact code" (checked), "Identifier Names Generator" (set to "hexadecimal"), "String Array" (checked), "Rotate String Array" (checked), "String Array Encoding" (set to "Off"), "String Array Threshold" (set to "0.8"), "Disable Console Output" (unchecked), "Debug Protection" (unchecked), "Sourcemaps" (set to "Off"), "Domain lock" (set to "domain.com"), and "Reserved Names" (containing "someVariable" and "RegEx").

A simple vista podemos apreciar que el código se ha vuelto ilegible y bastante más extenso de lo que originalmente era. Lo copiamos, pegamos en el archivo JS del proyecto (reemplazando el código anterior) y ejecutamos el documento HTML para ver si el resultado es el mismo. Si todo va bien, al pulsar el botón contenido en el documento HTML, el comportamiento de la página debe ser similar al que tenía con el código original.

The screenshot shows a browser window with the title "Ofuscación de código". Below the title, it says "Prueba" and "Esta página dice" followed by "Hola mundo!". There is a blue button labeled "Aceptar".

Análisis del proceso de ofuscación

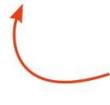
Como comentamos antes, si analizamos el código devuelto por el proceso de ofuscación, veremos que es mucho más extenso que el código original. ¿Por qué ocurre esto? El sistema de ofuscación se ocupa de leer aquellos datos expresados como variables, leyendas y funciones, entre otros, y transformar sus caracteres ASCII en un formato hexadecimal. Esto ya nos da un indicio de por qué el código se hace mucho más extenso.

Ajuste fino

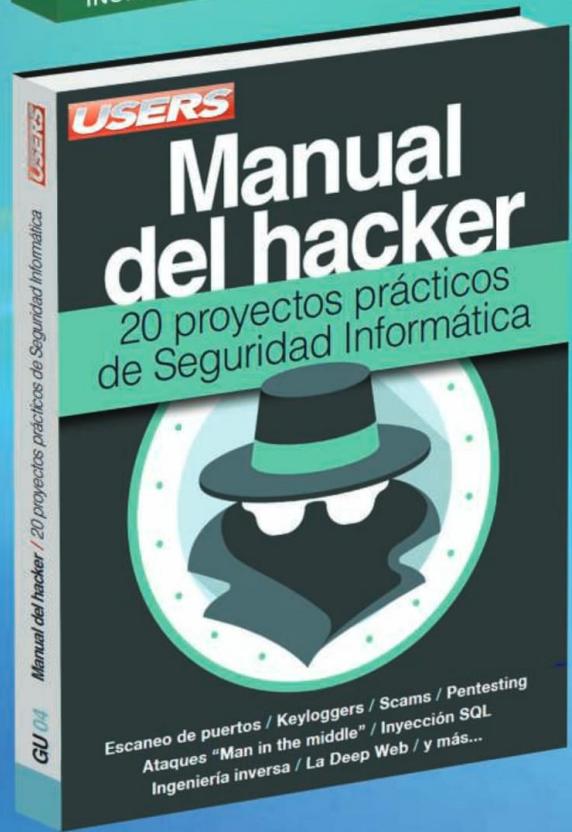
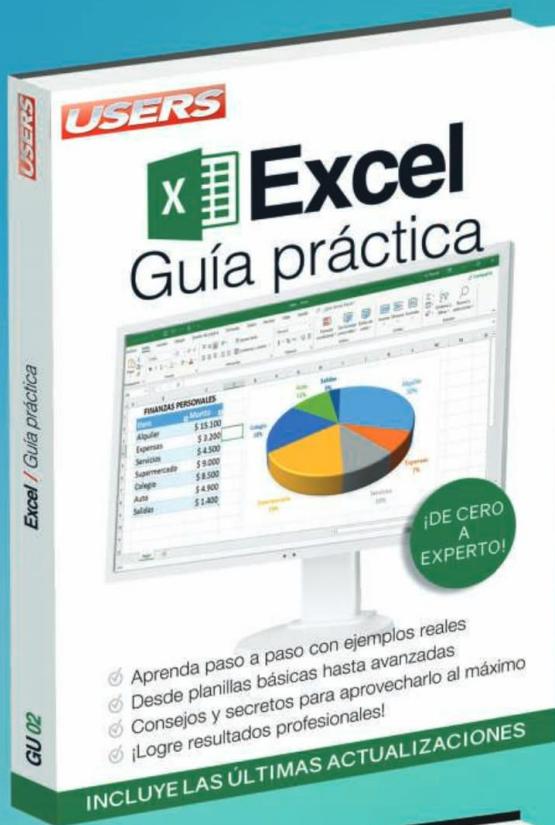
La plataforma online <https://obfuscator.io> nos permite realizar ajustes sobre el proceso de ofuscación, de manera tal que esto no sea un estándar definido solamente por ellos y para que también podamos optimizar la performance de un código ofuscado, según nuestra necesidad. Si miramos el sitio web en cuestión, por debajo del módulo de ofuscación, encontraremos una serie de **herramientas de configuración** que nos

permiten personalizar el modo de modificar el código fuente de nuestro proyecto. A través de este apartado, podemos realizar una serie de ajustes que nos permitirán optimizar el comportamiento del código JS ofuscado. Veamos a través de la siguiente tabla cuáles son las opciones más importantes para tener en cuenta al momento de ofuscar código:

AJUSTE	SIGNIFICADO
Reserved Names	Deshabilitar la ofuscación y la generación de identificadores, los cuales deben coincidir con patrones de expresiones. Esto impedirá que, si una variable determinada contiene el símbolo ^ por delante, no será procesada y deformada durante la ofuscación.
Domain lock	Al proceso de ofuscación de código se le agrega una determinada validación que verifica el DOMINIO en el cual se está ejecutando. Esto significa que, por más que alguien detecte y robe parte de un código JS, cuando lo ejecute en otro dominio diferente de WWW.MISITIO.COM, el código ofuscado no responderá a su procesamiento.
Encode String literals	Todos los textos literales serán convertidos previamente utilizando Base64 o RC4 y, para el proceso de decodificación, se aplicará una función especial que los devuelva a su modo original. Esto debe ser utilizado con cautela, ya que puede disminuir entre 30 y 35% la performance del código.
Disable console output	Al indicar esta opción, todo el código JS que tenga líneas que envían un resultado a la consola del navegador serán eliminadas. Esto evita que cualquier persona que aplique un intento de ingeniería inversa sobre nuestro código vea valores devueltos por consola, aun si estos son inyectados por la fuerza.
Debug protection	Marcar esta opción implica activar un intervalo de tiempo prolongado, que impida ver en tiempo real cualquier depuración que alguien realice sobre nuestro sitio web. De tal manera, seguir el proceso de nuestro código intentando hacer ingeniería inversa sobre él demandará un tiempo por demás molesto a quien se tome el trabajo.



En la tabla resumimos algunas de las funciones más importantes para tener en cuenta cuando realicemos la ofuscación de código JS. En el mismo sitio de **obfuscator**, hay un FAQ y un detalle mucho más amplio de lo mencionado. A su vez, los puntos más relevantes que pueden causar una caída en la performance del código ofuscado son mencionados para nuestro conocimiento y/o aceptación.



SUSCRIBETE

usershop.redusers.com

+54-11-4110-8700

usershop@redusers.com

JavaScript

Aprende a programar en el lenguaje de la Web

Configura el entorno de trabajo. Explora la sintaxis del lenguaje. Desarrolla desde ejercicios sencillos hasta proyectos complejos para aprender a programar en JavaScript.

En esta obra encontrarás todo lo necesario para dominar JavaScript, con explicaciones paso a paso, códigos de ejemplo e instrucciones detalladas.

Entre otros temas, aprenderás:

INTRODUCCIÓN

- Qué es JavaScript
- Configurar el entorno de trabajo
- VSCode
- JS integrado versus JS independiente
- JavaScript fuera del HTML
- Sintaxis básica de JavaScript
- Manejo de condicionales IF-ELSE

CONCEPTOS INICIALES

- Introducción a la Programación Orientada a Objetos
- Fechas, intervalos y cronómetros
- Arreglos
- Cadenas de texto
- Formularios y datos

PROYECTOS

- Controlar el DOM
- Hacer hablar a JavaScript
- Utilizar la API de notificaciones
- Almacenar datos localmente
- Implementar SQL offline
- Encriptar contenido
- Detectar conectividad a Internet
- Adaptar gráficos y multimedia según la performance
- Capturar fotografías y videos
- Acceder al hardware de los dispositivos
- Los sensores de movimiento
- Manejo de datos remotos con JSON
- Proteger el código mediante ofuscación



SOBRE EL AUTOR

Fernando Luna

es Analista de sistemas y Technical Writer. Lleva 25 años desarrollando software y escribiendo publicaciones orientadas a la programación. Cursa actualmente el Profesorado de Informática para oficializar su pasión por enseñar a programar.

UN MANUAL
PRÁCTICO PARA
APRENDER A
PROGRAMAR EN
JAVASCRIPT

NIVEL DE USUARIO
Intermedio / Avanzado

CATEGORÍA
Programación



REDUSERS.com

En nuestro sitio podrá encontrar noticias relacionadas y también participar de la comunidad de tecnología más importante de América Latina.

ISBN: 978-987-4958-08-2



9 789874 958082 >

¡DE CERO A EXPERTO!