

# Módulo 2. Herramientas avanzadas de Javascript

## Introducción

En este módulo se estudian conceptos esenciales, como son los eventos. Trabajaremos atributos y eventos on, addEventListener, y las categorías donde estos se aplican: *mouse*, teclado, formularios y navegador web. Luego, trataremos el almacenamiento de datos local y cómo integrarlo con DOM para leer y almacenar datos, además de cómo gestionar datos dinámicos.

Se introduce a promesas JavaScript (JS) y al comportamiento de las tecnologías de servidor. De esa forma, veremos cómo JS intercambia información a través de archivos en formato JSON o de servidores de *backend* reales.

## Video de inmersión

### Unidad 1: Eventos y almacenamiento de la información

En esta unidad se aborda el manejo de eventos como un aspecto fundamental en la programación, ya que permite que la aplicación responda a las acciones de los usuarios, como hacer clic en botones o enviar formularios. Los eventos son los desencadenantes de la interactividad en una aplicación web.

Además, se explora la capacidad de los navegadores web para almacenar datos localmente. Esto resulta útil para guardar preferencias y configuraciones, mejora la velocidad de respuesta y reduce la carga en el servidor *backend*, al minimizar las peticiones que este recibe.

#### Tema 1: El manejo de eventos

En JavaScript, los eventos permiten controlar las acciones realizadas por los usuarios en aplicaciones web. Podemos detectar diferentes eventos y definir acciones específicas en respuesta a ellos.

Por ejemplo, cuando se hace clic en un botón o en un elemento de menú, se puede enviar un formulario o navegar a otra sección de la aplicación. Existen muchos eventos en JavaScript, pero la mayoría son aplicables a cualquier elemento HTML, lo que facilita su aplicación. Categorizar los eventos ayuda a organizarlos de manera efectiva y facilita recordarlos.

#### Cómo manejar eventos en JS

En el manejo de eventos en JavaScript se enlaza un objeto JS con un elemento HTML para asociar una función a un evento específico. Cuando el evento ocurre, se ejecuta la función asociada. Esto se conoce como manejo de eventos o *event handling*. Las funciones pueden ser convencionales o anónimas, creadas exclusivamente para cada evento.

El navegador web tiene un escuchador que constantemente verifica si hay funciones asociadas al evento detectado. Si existe una función, se invoca como respuesta. Este proceso se conoce como *callback*.

#### Formas de utilizar eventos

Actualmente, JavaScript cuenta con diferentes formas de manejar eventos. Todas llevan al mismo resultado, pero se aplican de manera distinta. Veamos, en el siguiente gráfico, cuáles son estas formas.

Figura 1: Manejar eventos en JS

## Manejar eventos en JS



Fuente: elaboración propia.

### Eventos: qué sí y qué no

En el gráfico anterior se muestran los tres tipos de eventos que siguen vigentes en JavaScript. Se desalienta el uso del primero de ellos, atributos ON, desde 1999, cuando la organización W3C decidió separar el código de HTML, CSS y JavaScript en archivos diferentes.

Tener el código correctamente separado ayuda a que el mantenimiento del software sea más efectivo, y a evitar la repetición innecesaria de bloques de código.

El uso de eventos ON sigue vigente actualmente, pero en el ámbito laboral se alienta el uso de métodos que escuchen eventos (opción 3). Esta es la forma más moderna que llegó a JavaScript en 2015 y, como forma parte del estándar ES6, es la más requerida.

Los avances en esta lectura se enfocarán en esta última opción: **addEventListener**.

### Tipos de eventos

Categorizar los diferentes eventos ayuda a ordenar la forma en la cual recordaremos su uso. A continuación, se muestra una manera óptima de ordenarlos.

Figura 2: Categoría de eventos en JavaScript



Fuente: elaboración propia.

### Eventos del mouse

Estos eventos están asociados a todo aquel evento disparado por el puntero del mouse, e incluso por el tap que realizamos sobre elementos HTML, a través de pantallas táctiles. Estos se subdividen en los siguientes tipos.

Tabla 1: Referencia de los eventos del mouse

Evento del mouse	Descripción
Click	Cuando el usuario realiza un clic con el mouse en un elemento.
Dblclick	Cuando el usuario realiza doble clic con el mouse en un elemento.

<i>Mousedown</i>	Cuando el usuario pulsa el botón sobre un elemento y lo mantiene pulsado (el inicio del evento clic).
<i>Mouseup</i>	Cuando el usuario suelta el botón que pulsó sobre un elemento (el fin de un evento clic).
<i>Mouseover</i>	Cuando el usuario posiciona el <i>mouse</i> sobre un elemento HTML.
<i>Mouseout</i>	Cuando el usuario quita el <i>mouse</i> de arriba de un elemento HTML.
<i>Mousemove</i>	Cuando el usuario mueve el <i>mouse</i> sobre un elemento HTML.

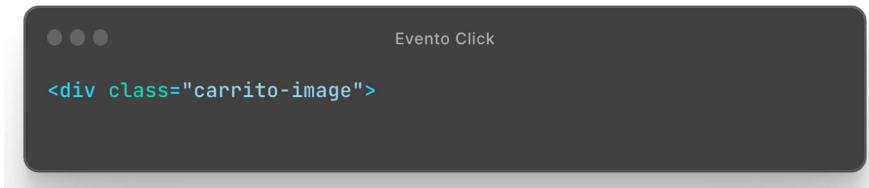
Fuente: elaboración propia.

A continuación, se elaboran algunos ejemplos de estos eventos.

### Evento clic

Disponemos de un proyecto donde tenemos un div con una clase CSS denominada carrito-image. Este contiene una imagen representativa de un carrito de compras de una aplicación de e-commerce.

**Figura 3: Evento clic**

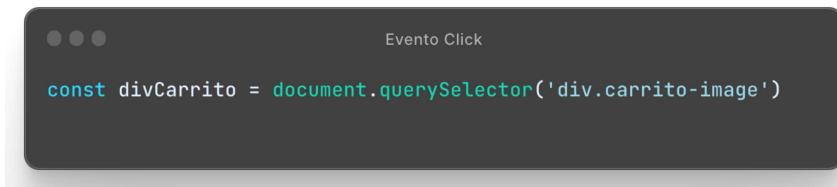


```
<div class="carrito-image">
```

Fuente: elaboración propia.

Desde JavaScript debemos enlazarnos al tag `<div>`, y definir una función que nos lleve al documento HTML correspondiente al carrito.

**Figura 4: Evento clic. Enlace a <div>**

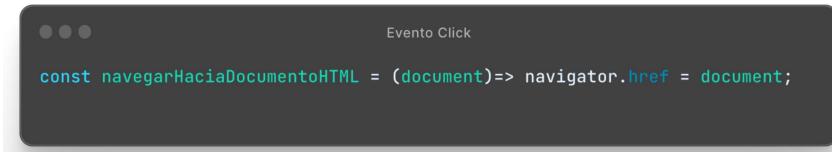


```
const divCarrito = document.querySelector('div.carrito-image')
```

Fuente: elaboración propia.

Disponemos de una función JS genérica que nos permite definir la navegación web por todos los documentos HTML de nuestra aplicación web. Esta función recibe como parámetro el nombre del documento o ruta HTML a la cual debemos navegar, y ejecuta este proceso mediante el objeto **navigator**, nativo de JavaScript.

**Figura 5: Evento clic. Objeto navigator**



```
const navegarHaciaDocumentoHTML = (document)=> navigator.href = document;
```

Fuente: elaboración propia.

Con todo este proceso correctamente definido, solo nos queda aplicar el evento clic en el <div> correspondiente, todo esto se maneja siempre desde JavaScript. Para ello, se utiliza el método **addEventListener**. Como se mencionó anteriormente, el método espera un primer parámetro denominado “evento”, y un segundo parámetro denominado “callback”.

### Funciones callback

El *callback* está definido por una función anónima, o *arrow function*, que puede llamar a otra función general de la aplicación, o contener un bloque de código específico correspondiente a la tarea a ejecutar.

Figura 6: Evento clic. Función *callback*



Fuente: elaboración propia.

En este caso, llamamos a la función de navegación JS predeterminada. Esta es genérica y reutilizable en cualquier otra parte de la aplicación. Es la forma más apropiada y buscada en la actualidad para controlar los diferentes eventos que acontecen en un documento HTML, desde el código JS.

### Evento *mousemove*

Veamos otro ejemplo más. En este caso, se aplica un evento sobre la acción de mover o posicionar el *mouse* sobre un elemento HTML. Realizaremos una pequeña modificación del puntero del *mouse*, al cambiar el cursor sobre el hipervínculo por el dedo señalador.

Este evento lo maneja HTML, pero como este no posee un hipervínculo aquí, debemos controlarlo con JavaScript. Veamos cómo realizarlo.

Figura 7: Evento clic. Evento *mousemove*



Fuente: elaboración propia.

Al agregar el evento **mousemove** sobre el elemento **divCarrito**, definimos, en la función *callback* asociada a este evento, que se utilice la propiedad *style* de JS. En esta se encontrará la propiedad denominada *cursor*, que dispone de diferentes opciones para cambiar el puntero del mouse. *Pointer* es el nombre del puntero que hace referencia a la mano que señala en los hipervínculos.

De esta forma, activaremos el evento *mousemove* cuando el usuario posicione el cursor del *mouse* sobre **divCarrito**. El cursor cambiará, para notificar visualmente que ese elemento HTML posee un hipervínculo.

### Eventos del teclado

Los eventos de teclado están regidos por todo aquello que refiere a la pulsación de teclas y al cambio de estados de determinados elementos. Para poder aplicar o controlar este tipo de eventos, debemos pensar en elementos HTML que nos permitan el ingreso de información mediante INPUT; se utiliza, por supuesto, un teclado o el portapapeles del sistema.

**Tabla 2: Referencia de los eventos del teclado**

Evento de teclado	Descripción
Keypress	Cuando el usuario pulsa las teclas del teclado, usualmente mientras escribe en un campo de texto.
Keydown	Cuando el usuario pulsa una tecla y la mantiene pulsada, para escribir en un campo de texto (el proceso inicial del evento <i>keypress</i> ).
Keyup	Cuando el usuario suelta la tecla pulsada en el proceso de escritura en un campo de texto (el proceso final del evento <i>keypress</i> ).
Change	Detecta algún cambio acontecido sobre algún elemento HTML. No precisamente sobre un elemento del tipo <i>input</i> (por ejemplo, cambiar la imagen mostrada en un tag <i>&lt;img&gt;</i> disparará el proceso <i>change</i> ).
Search (*)	Este evento está reservado al elemento HTML <i>&lt;input type="search"&gt;</i> . Un <i>input type</i> que permite la escritura de texto, pero 100 % enfocado en poder realizar búsquedas de contenido. Cuando el usuario escribe en este campo y pulsa la tecla <i>enter</i> , se dispara el evento “ <i>search</i> ”.

Fuente: elaboración propia.

Los elementos HTML del tipo ***input*** serán los protagonistas de este proceso de manejo.

### Evento ***keypress***

Para capturar el evento *keypress*, podemos imaginar que disponemos de un formulario web donde el usuario se ocupa de cargar contenido, por ejemplo: dar de alta nuevos productos. Este formulario está conformado por varios campos del tipo *input*, que definen los datos que debemos cargar.

Veamos un ejemplo del formulario web, a través del siguiente gráfico, para contextualizar.

**Figura 8: Ejemplo de formulario web**

# Ventadefrutas.com

## Ingresar nuevos productos

Fuente: elaboración propia.

Todos los *input types* están enlazados desde JS mediante el método **querySelector**. También disponemos de un objeto literal denominado **nuevoProducto**.

**Figura 9: QuerySelector**

```
... Evento change

const btnGuardar = document.querySelector('button#btnGuardar')

const inputCodigo = document.querySelector("input#inputCodigo")
const inputNombre = document.querySelector("input#inputNombre")
const inputImporte = document.querySelector("input#inputImporte")
const inputStock = document.querySelector("input#inputStock")

const nuevoProducto = {
  id: 0,
  nombre: '',
  importe: 0.00,
  stock: 0
}
```

Fuente: elaboración propia.

Si, por ejemplo, deseamos capturar el evento *keypress* de **inputNombre**, entonces debemos referenciar el método *addEventListener* sobre este elemento. Esto se hace como se muestra a continuación.

**Figura 10: Evento keypress**

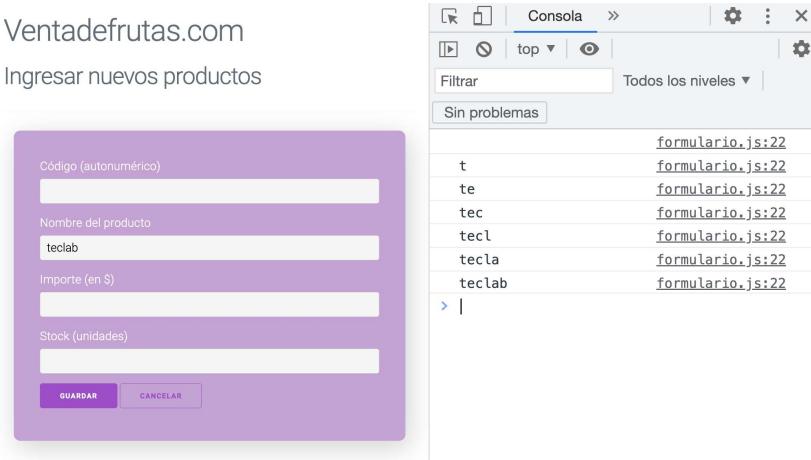
```
... Evento keypress

inputNombre.addEventListener("keypress", ()=> {
  console.log(inputNombre.value)
})
```

Fuente: elaboración propia.

De esta forma, cada vez que se tipee en el campo de formulario en cuestión, podremos ver en la consola JS lo que se escribe.

**Figura 11: Resultado del funcionamiento del evento keypress**



Fuente: elaboración propia.

Cada vez que se tipea en el campo de formulario, **console.log** refleja en la consola JS el texto almacenado en este, y detecta el evento keypress en cuestión.

### Evento *change*

De acuerdo con el ejemplo anterior, nuestro objetivo principal es capturar el evento *change* de cada *input type* (con excepción de *inputCodigo*); y guardar el contenido escrito en el formulario web, en cada propiedad homónima que conforma el objeto literal **nuevoProducto**.

Este evento será disparado en el momento en el que el usuario deje de escribir contenido en el *input type* y este pierda el foco. Allí se inicia el evento *change*, que valida si hubo cambios en el estado del *input type* en cuestión. De haber cambios, dispara el evento en sí y, por lo tanto, su *callback* definido como acción.

**Figura 12: Evento *change***

```
Evento change

inputNombre.addEventListener("change", ()=> {
    nuevoProducto.nombre = inputNombre.value
})
inputImporte.addEventListener("change", ()=> {
    nuevoProducto.importe = inputImporte.value
})
inputStock.addEventListener("change", ()=> {
    nuevoProducto.stock = inputStock.value
})
```

Fuente: elaboración propia.

De esta manera, y sin esperar que el usuario pulse el botón **guardar**, conformamos la estructura del objeto literal **nuevoProducto**, para que esté listo para ser guardado en una aplicación *backend*, o que se sume como nuevo objeto en un *array* dedicado.

**Figura 13: Validación de funcionamiento del evento *change***

Ventadefrutas.com

Ingresar nuevos productos

(índice)	Valor
id	0
nombre	"Notebook ultraliv...
importe	245600
stock	35

Fuente: elaboración propia.

## Eventos de formulario

Los formularios web existen desde mucho antes de la llegada de JS a los navegadores web. Recopilan información a través de campos de texto, y la envían a través de solicitudes HTTP. Todo su comportamiento y funcionalidad está supeditada por el intérprete de HTML del motor de los navegadores web, por lo que prescinde por completo del lenguaje JavaScript.

Más allá de esto, JavaScript posee mecanismos para controlar el comportamiento de un formulario web, para ello utiliza eventos dedicados. Entre los eventos específicos de formularios, encontramos los siguientes.

**Tabla 3: Referencia de los eventos del teclado**

Evento de formulario	Descripción
<i>Focus</i>	Cuando el usuario envía el foco a un elemento del formulario, por ejemplo: posiciona el cursor del mouse en un <i>input type</i> al hacer clic, o mediante la tecla de tabulación.
<i>Blur</i>	Opuesto a <i>focus</i> . Cuando el elemento del formulario pierde el foco.
<i>Submit</i>	Cuando se envía un formulario. Es disparado inicialmente por el elemento <i>input type submit</i> o <i>button type submit</i> .

Fuente: elaboración propia.

Los eventos *focus* y *blur* son comúnmente utilizados para cuestiones muy puntuales o personales. Sin embargo, su comportamiento puede ser totalmente definido desde CSS, a través de las seudoclases apropiadas, por lo cual dejaremos esta tarea del lado de CSS.

En cuanto al evento *submit*, este sí suele ser mayormente controlado desde JS, dado que la interacción actual de formularios generalmente trabaja con servicios web de *backend*. Estos difieren ligeramente del funcionamiento de aplicaciones de servidor convencionales, pensadas para controlar formularios web.

### El evento *submit*

Este evento se produce cuando pulsamos el botón *submit* de un formulario web. Lo podemos detectar desde JS y frenar su comportamiento predeterminado, al tomar el control del formulario HTML y manejar su lógica desde JS. Para lograr esto, dependemos de un objeto global denominado **event**, que forma parte del Core JS. Este objeto global está siempre presente y se puede evocar de manera instanciada dentro de la función *callback*, que se define en los eventos que escuchamos.

**Figura 14: Evento submit**

```
Evento submit

const formulario = document.querySelector("form")

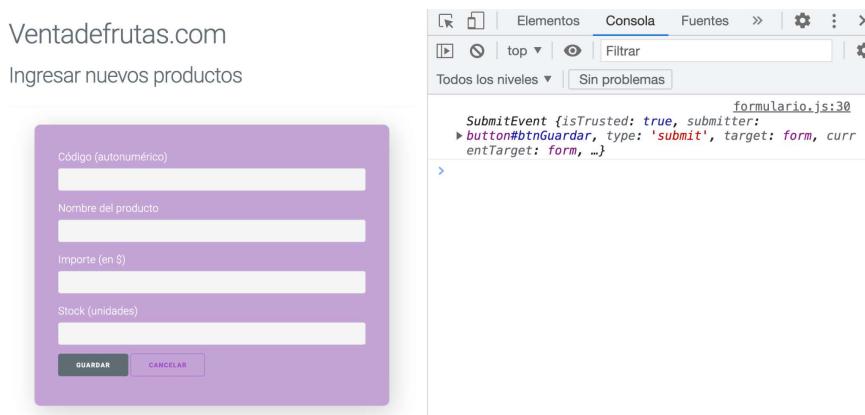
formulario.addEventListener("submit", (e)=> {
    console.log(e)
})
```

Fuente: elaboración propia.

**Button type submit** dispara el evento, sin embargo, este evento es controlado en sí por el *tag form*. Por lo tanto, debemos enlazarnos desde JS a dicho *tag HTML <form>*, para poder escuchar el evento en cuestión. Luego de enlazarnos, definimos el objeto global Event en la función de *callback* del evento *submit*.

Usualmente se lo representa bajo los siguientes sinónimos: e, evt, ev, event. Elegimos el primero para representarlo. Al probar el evento *submit*, veremos que en la consola JS se muestra la información del objeto global Event.

**Figura 15: Al pulsar el botón Guardar, generamos el evento submit**



Fuente: elaboración propia.

### Prevenir el comportamiento por defecto

Al disponer del objeto global **Event**, podemos hacer uso de un método interno denominado **preventDefault()**. Al referenciar este método, obligamos desde JS que el navegador web no ejecute de forma predeterminada el comportamiento del formulario web. Así podremos tener control del envío de los datos de este formulario, y manejar la lógica desde JS. A continuación, veamos un ejemplo de código.

**Figura 16: PreventDefault()**

```
Evento submit

formulario.addEventListener("submit", (e)=> {
    e.preventDefault()

    inputCodigo.value = obtenerCodigo()
    nuevoProducto.id = nuevoProducto.value
    console.table(nuevoProducto)
})
```

Fuente: elaboración propia.

Mediante `e.preventDefault()`, controlamos el comportamiento predeterminado del formulario. Invocamos a una función con retorno, que genera el ID automático para cada nuevo producto, y asienta el número correspondiente en `input type codigo`.

Por último, asentamos el código numérico generado para nuestro producto, en la propiedad `id` del objeto `nuevoProducto`, y enviamos el objeto literal a la Consola JS.

Figura 17: Cómo se visualiza en la Consola JS un producto cargado en el formulario web

Ventadefrutas.com  
Ingresar nuevos productos

Código (autonumérico)  
5802

Nombre del producto  
Notebook Ultraliviana 17

Importe (en \$)  
245600

Stock (unidades)  
42

GUARDAR CANCELAR

Consola

Todos los niveles Sin problemas

formulario.js:34

(índice)	Valor
id	5802
nombre	'Notebook Ultraliviana 17'
importe	245600
stock	42

Fuente: elaboración propia.

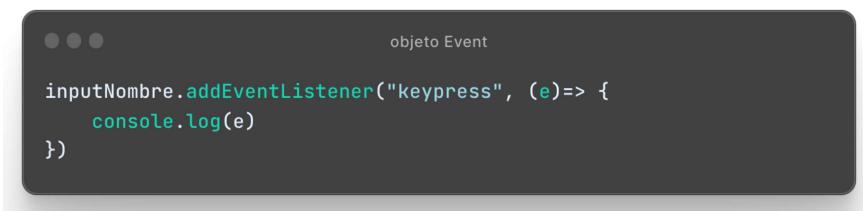
Con el objeto global Event, al controlar el comportamiento por defecto del evento `submit` de un formulario web, también tenemos el control para evitar que al ocurrir este evento, el formulario web se resetee de forma predeterminada.

En el gráfico anterior podemos ver cómo el producto cargado en el formulario web es visualizado en la Consola JS, con la información almacenada en el objeto `nuevoProducto`.

### Información adicional de un evento

A continuación, podemos ver otro escenario donde el objeto global Event es capaz de brindar información adicional sobre diferentes eventos en cuestión. Para ello, volveremos por un momento al uso de los eventos de teclado.

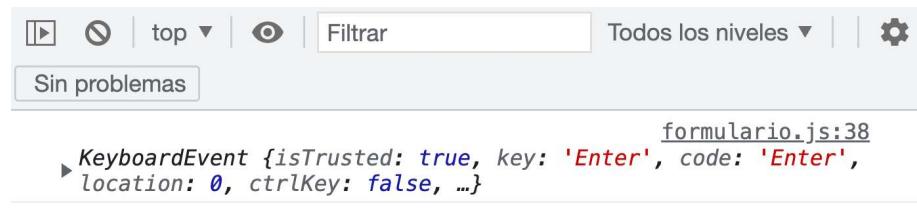
Figura 18: Objeto Event



Fuente: elaboración propia.

Si definimos, sobre cualquier campo de formulario, el evento `keypress`, y adjuntamos el objeto global Event como parámetro en la función `callback`, podemos ver que el mismo tiene la capacidad de, por ejemplo, identificar cuál elemento HTML genera el evento en cuestión. Esto se denomina información del evento.

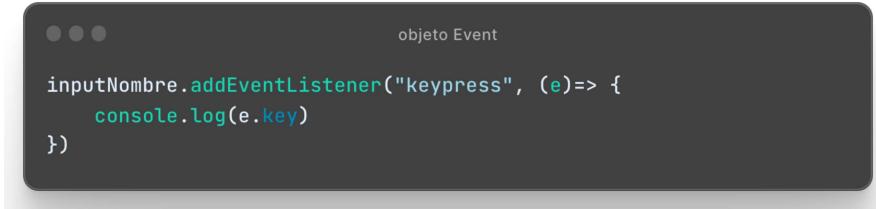
Figura 19: Ejemplo de código



Fuente: elaboración propia.

En el ejemplo de código anterior, podemos ver cómo el objeto global Event envía a la consola JS la información completa del elemento HTML que generó el evento. En esa información podemos identificar datos clave que pueden ser de utilidad, como por ejemplo, qué tecla fue pulsada.

**Figura 20: Propiedad key**



Fuente: elaboración propia.

La propiedad *key* del objeto global Event devuelve el nombre de la tecla pulsada. Esto nos ayudará a identificar qué tecla se pulsa y asociar un evento específico a la misma. (Ejemplo: el usuario pulsó *enter*, entonces envía el foco al siguiente campo de formulario).

#### Propiedades adicionales que podemos leer

Si queremos detectar una combinación de teclas como evento, podemos identificar las siguientes teclas extendidas.

**Tabla 4: Controlar teclas extendidas con el objeto global Event**

Propiedad	Descripción
CtrlKey	Al retornar su valor <i>true</i> , indica que la tecla CTRL estaba pulsada.
AltKey	Al retornar su valor <i>true</i> , indica que la tecla ALT estaba pulsada.
ShiftKey	Al retornar su valor <i>true</i> , indica que la tecla SHIFT estaba pulsada.
metaKey	Al retornar su valor <i>true</i> , indica que la tecla win (en Windows) o la tecla command (en Mac OS) estaba pulsada.

Fuente: elaboración propia.

Esta es la fórmula que utiliza, por ejemplo, el cliente web GMail para poder definir la combinación de teclas para crear un nuevo *mail*, enviar un *mail* ya escrito, etcétera.

También tenemos disponible el acceso a los atributos del elemento HTML que generó el evento, a través de *e.target*. Por ejemplo, podemos identificar el ID de un elemento HTML y asignarle un evento dedicado que se dispare ante determinada acción del usuario.

**Figura 21: E.target**

```

    objeto Event

button.addEventListener("keypress", (e)=> {
    if (e.target.id === 'imgCarrito') {
        navigator.href = 'carrito.html';
    }
})

```

Fuente: elaboración propia.

## Eventos de navegador

El objeto global JS **window** brinda información sobre diferentes eventos que ocurren en el navegador web, y que están por fuera del manejo predeterminado que pueda tener JS. A continuación, se enumeran los eventos de navegador principales que podemos controlar.

**Tabla 5: Controlar teclas extendidas con el objeto global Event**

Propiedad	Descripción
<i>Scroll</i>	Detectar cuando el usuario hace <i>scroll</i> sobre un documento HTML.
<i>Resize</i>	Detectar que se redimensiona la ventana de navegación.
<i>Hashchange</i>	Detectar si cambió la URL, definida por el carácter # (utilizado en SPA).
<i>Load</i>	Detectar cuando se carga un documento HTML.
<i>Unload</i>	Detectar cuando se descarga o se cierra un documento HTML.
<i>DOMContentLoaded</i>	Detectar cuando se cargó un documento por completo (tags HTML, archivos de recursos externos como CSS, JS, imágenes, etc.).
<i>Online</i>	Detectar cuando retorna la conectividad a internet.
<i>Offline</i>	Detectar cuando se pierde la conectividad a internet.

Fuente: elaboración propia.

Como podemos ver, los eventos son totalmente controlables por JavaScript, y el mecanismo del método `addEventListener` es el que nos ayudará a definir qué evento ejecutar y sobre qué elemento HTML definirlo.

Existen otros eventos JS controlables, que están asociados a elementos específicos, tales como elementos HTML `<audio>` y `<video>`. En ellos podemos tener un dominio total de los controles de reproducción multimedia.

### Actividad 1

Descargar el siguiente archivo de proyecto comprimido en formato .ZIP, descomprimirlo y abrir la carpeta de proyecto con VS Code.



Editar el archivo JavaScript. En el mismo, se encuentran referenciados los principales elementos del formulario HTML.

Definir en este formulario el evento `change`, explicado a lo largo de esta lectura, para cada uno de los `input types` del formulario

(excepto `input type` `codigo`).

Agregar la función `callback` a cada evento `change`, para que se complete la propiedad correspondiente al `input type`, en el objeto `nuevoProducto`, tal como se explica en la lectura.

Luego de finalizar estos pasos, ejecutar el documento HTML en el navegador web Chrome. Utilizar siempre **live server**. Abrir Developer tools e ir a la pestaña Consola JS.

Probar la carga de contenido en cada campo de formulario, pasando entre uno y otro mediante la tecla TAB del teclado.

Una vez completados los campos en cuestión, verificar en la consola JS si el objeto `nuevoProducto` posee todas las propiedades con datos válidos recuperados del formulario web. Para esto, ejecutar `console.table(nuevoProducto)`, que permite ver cómodamente la información del objeto literal.

Por último, agregar el enlace JS al elemento `<form>`, con `querySelector`, y capturar luego el evento `submit` de este; prevenir su comportamiento por defecto y ejecutar la cláusula `console.table()` mencionada anteriormente, a través de dicho evento.

**Para conocer la resolución del caso, descargá el archivo al final de la lectura.**

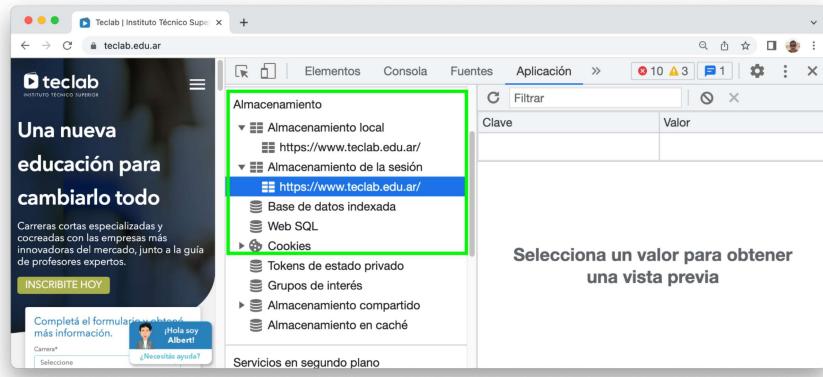
## Tema 2: Datos: transporte y almacenamiento

WebStorage es una tecnología disponible en todos los navegadores web. Permite a las aplicaciones web almacenar datos en el navegador de los usuarios. Estos datos pueden ser temporales o persistentes, lo que significa que permanecen disponibles incluso después de cerrar y volver a visitar la página web. Esto brinda a los desarrolladores una forma conveniente de guardar y recuperar información en la máquina del usuario, al mismo tiempo que mejora la experiencia del usuario y reduce las interacciones con el servidor *backend*.

### Almacenamiento de datos en el cliente web

Desde la llegada de HTML5 en su versión final en 2014, el almacenamiento web del lado del cliente (en su *web browser*), dejó de estar regido por las *cookies* para pasar a disponer de varias opciones. Entre estas opciones encontramos: LocalStorage, SessionStorage, e IndexedDB, entre otras no tan funcionales como WebSQL.

**Figura 22: Formas de almacenamiento local en un navegador web**



Fuente: elaboración propia.

Toda la información de almacenamiento de datos en un navegador web se encuentra en el apartado aplicación (*application*, en inglés), en DevTools. Allí podremos ver todas las opciones mencionadas, e incluso acceder a una parte de los datos que estén almacenados.

Cada información que se guarda allí se relaciona directamente con el sitio o aplicación web que generó estos datos. La misma

no es *cross-site*, por lo tanto, no podremos compartir información entre diferentes sitios o aplicaciones web.

## Tipos de almacenamiento local

En la siguiente tabla vemos los diferentes tipos de *webstorage* y sus características.

Tabla 6: Descripción de los tipos de almacenamiento local (*webstorage*)

Webstorage	Descripción
<b>LocalStorage</b>	Posee la capacidad de almacenar entre 5 y 10 MB de información por dominio, según el navegador web o el tipo de dispositivo. Funciona bajo una estructura de pares clave-valor, y es persistente hasta que la aplicación web o el usuario eliminan los datos allí almacenados.
<b>SessionStorage</b>	Posee las mismas características de almacenamiento y estructura que LocalStorage. Se diferencia de este porque la información es almacenada por sesión (de forma temporal). Cuando la sesión finaliza (el usuario cierra la pestaña), lo almacenado se pierde.
<b>IndexedDB</b>	Su capacidad de almacenamiento es significativa. Se estima que puede ocupar hasta el 80 % del espacio del dispositivo. IndexedDB posee un formato del tipo base de datos NoSQL, para aprovecharla al máximo requiere un manejo muy avanzado de JS. Los datos persisten de la misma forma que con LocalStorage.
<b>WebSQL</b>	Inicialmente, fue un excelente mecanismo de almacenamiento local basado en el lenguaje SQL ( <i>database</i> , tablas, campos, registros). Desde 2010 dejó de evolucionar. Muchos navegadores web actualmente desactivan WebSQL y, en algunos años, se retirará definitivamente su soporte completo.
<b>Cookies</b>	Las <i>cookies</i> existen desde que nacieron los navegadores web. Poseen muy baja capacidad de almacenamiento, 4 KB. Desde hace unos años se desalienta su uso. Es probable que en los próximos años desaparezcan como opción en los navegadores web, al igual que WebSQL.

Fuente: elaboración propia.

## LocalStorage y SessionStorage

El mecanismo de uso de ambos es similar. Están integrados como un objeto global del Core de JavaScript, dentro del objeto **window**, y pueden aprovecharse al máximo en desarrollos web *frontend*. Poseen cuatro métodos simples para manipular la información a grabar, leer o eliminar de *webstorage*. Veamos a continuación cuáles son.

### setItem()

El método **setItem()** recibe dos parámetros: **clave** y **valor**. La clave corresponde al identificador amigable con el cual queremos definir la información a almacenar.

Por ejemplo, si deseamos almacenar el nombre del usuario que accede a nuestra plataforma, para darle la bienvenida a través

de un mensaje en pantalla, podemos crear una clave y almacenar su nombre como valor.

**Figura 23: setItem()**



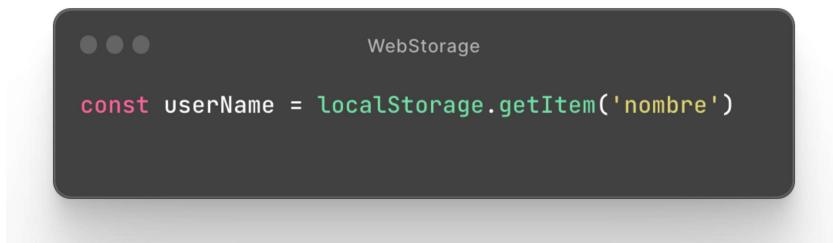
Fuente: elaboración propia.

Luego, cada vez que el usuario acceda a la aplicación web, leemos este valor desde LocalStorage y armamos un mensaje personalizado de saludo, que se muestra en algún lugar del documento HTML.

**GetItem()**

Este método permite recuperar un valor o clave almacenado previamente en WebStorage. En este caso, el método recibe solo un parámetro, que corresponde al nombre de la clave a recuperar.

**Figura 24: GetItem()**



Fuente: elaboración propia.

Este método funciona con retorno implícito, por lo tanto, debemos declarar una variable o constante para poder recuperar el valor almacenado, si es que existe. En el caso de que no exista, el valor de la variable o constante será **null**.

**Figura 25: Definir valor alternativo**



Fuente: elaboración propia.

Ante esta última situación, podemos utilizar el operador lógico OR para definir un valor alternativo y que la variable o constante no obtenga un valor inconsistente para la lógica de nuestra aplicación.

**RemoveItem()**

Finalmente, ante la necesidad de remover o eliminar una **clave - valor** que ya no necesitamos, podemos recurrir a este otro

método. Se informa como parámetro el nombre de la clave a eliminar.

**Figura 26: RemoveItem()**

The screenshot shows a dark-themed browser developer tools console window titled "WebStorage". It contains a single line of code: `localStorage.removeItem('nombre')`. The code is highlighted in green, indicating it is valid JavaScript.

Fuente: elaboración propia.

### Clear()

Por último, si hacemos un uso intensivo de `webstorage` y por alguna situación específica necesitamos eliminar todo rastro de este, disponemos del método `clear()`, que se ocupa de eliminar toda **clave-valor** existente. En el caso de utilizar este último método, debemos tener precaución, ya que borrará todo sin previa confirmación del procedimiento.

Todos estos métodos son comunes a `LocalStorage` y `SessionStorage`, con la salvedad de las diferentes formas en las que se comporta cada uno de ellos.

### El objeto global JSON

El uso de `LocalStorage` o `SessionStorage` para almacenar información nos limita a guardar tipos de datos en formato `string`. Para datos simples, como el almacenamiento de un nombre (que pusimos como ejemplo), no habrá ningún problema. Habrá inconvenientes si tenemos la necesidad de almacenar datos en un formato de objeto.

**Figura 27: Ejemplo de código. Creación de clave denominada nombre**

The screenshot shows a dark-themed browser developer tools console window titled "WebStorage". It contains two lines of code: `localStorage.setItem('nombre', 'Teclab')` and `const producto = {id: 123, nombre: 'Computadora', importe: 125_000}; localStorage.setItem('producto', producto)`. The first line is highlighted in green, and the second line is highlighted in blue, indicating they are valid JavaScript statements.

Fuente: elaboración propia.

A través de este último ejemplo de código, vemos la creación de una clave denominada `nombre`, con un valor del tipo `string` asociado. Luego, creamos un objeto literal llamado `producto`, con algunos datos simples. Al intentar almacenarlo en `LocalStorage`, veremos que la clave se crea sin problema, pero que su valor almacenado no es la estructura de objeto literal en sí.

**Figura 28: Dificultades en valor almacenado**

Clave	Valor
nombre	Teclab
producto	[object Object]

Fuente: elaboración propia.

Aquí comprobamos que **localStorage** solo acepta valores del tipo *string*. Si intentamos recuperar el valor almacenado en la clave “producto”, veremos un texto similar al del ejemplo gráfico; por lo tanto, será inservible para nuestra labor. Para resolver este inconveniente, existe el objeto global JSON, integrado al Core del lenguaje JavaScript.

## JSON

Este objeto nos permite tomar estructuras del tipo objeto literal, *array* de objetos literales, y *arrays* de elementos, y convertirlas a un formato que pueda ser almacenado, por ejemplo, en *webstorage*. Para lograr esto, este objeto tiene dos métodos.

### JSON.stringify()

Este método recibe por parámetro una estructura de objeto JS. Dicha estructura puede ser un objeto literal, *array* de objetos o *array* de elementos. El método *stringify* la analiza y se ocupa de convertir toda la estructura en una de tipo *string*. Esta última retorna como resultado.

Veamos un ejemplo, a continuación.

**Figura 29: JSON**



```
const producto = {id: 123, nombre: 'Computadora', importe: 125_000}
JSON.stringify(producto)
/*
    RESULTADO DE LA CONVERSIÓN:
    '{"id":123,"nombre":"Computadora","importe":125000}'
*/
```

Fuente: elaboración propia.

En el ejemplo anterior vemos cómo la estructura de objeto literal se mantiene, con la diferencia de que se agregan comillas simples y dobles sobre su estructura original. Esto hace que ahora sí se pueda almacenar como una estructura del tipo *string*.

Este mismo mecanismo se debe realizar para guardar estructura de datos en formato JSON, en una aplicación de *backend*. Los navegadores web (*frontend*) y los servidores web (*backend*) no pueden intercambiar objetos entre sí, solo pueden intercambiar estructuras de información en formato *string*.

### JSON.parse()

Este método realiza la operación inversa a **stringify()**. Recibe como parámetro una estructura del tipo *string*, que debe respetar la estructura de un objeto literal, *array* de objetos literales o *array* de elementos, para luego ocuparse de convertirla en un objeto

como tal.

El nombre, parse, es un término en inglés que se refiere a analizar sintácticamente. Por ello, se ocupa de analizar la estructura del tipo *string* para validar que esta pueda convertirse nuevamente en una estructura de objeto o array en JS.

**Figura 30: JSON.parse()**



Fuente: elaboración propia.

De lograr exitosamente su conversión, retornará un objeto como tal.

### Limitaciones

Dado que un objeto literal puede contener propiedades y métodos, estos últimos no deben formar parte del objeto literal que intentemos convertir mediante el método **stringify()**, porque este método solo convierte estructuras basadas en propiedades y valores.

Esta restricción aplica tanto para un objeto literal con métodos como también para un *array* de objetos literales con métodos, o un *array* de objetos creados a partir de una clase JS que contenga métodos internos.

### Actividad 2

Recuperar el proyecto de la actividad 1.

Identificar el evento **submit** definido en la actividad anterior. Justo debajo de donde se invoca **console.table()**, para ver el objeto **nuevoProducto**, definir una nueva línea de código.

En esa línea, agregar una nueva clave a LocalStorage llamada **MiProducto**, y almacenar como valor la información contenida en el objeto **nuevoProducto**.

Recordar: aplicar el uso del objeto global **JSON** para convertir a *string* el objeto literal, antes de guardarlo en LocalStorage.

**Para conocer la resolución del caso, descargá el archivo al final de la lectura.**

### Tema 3: Combinar herramientas

Los temas anteriores describen las características sólidas de JavaScript y su interacción con las aplicaciones *frontend*. Trabajamos con el DOM HTML, la interacción con elementos HTML simples y bloques HTML, y el control de eventos en un entorno web y navegador.

El paso siguiente es aplicar estos conceptos de manera práctica, para tener una comprensión más amplia de cómo implementar el ecosistema de herramientas de JavaScript en una aplicación *frontend*.

### Integración de DOM y datos

En esta unidad buscamos aplicar los conocimientos adquiridos mediante un ejemplo práctico. El objetivo es integrar el DOM HTML, datos dinámicos, arrays de objetos y eventos, así como la persistencia de datos de forma dinámica. Recomendamos utilizar Live server durante las pruebas, para garantizar una correcta visualización de la ejecución del ejemplo.

Descarguemos el proyecto llamado **Integración DOM y datos**, y editémoslo con VS Code. Luego, veamos su código base para entender este ejemplo práctico.



Figura 31: Código base del ejemplo práctico

The screenshot shows the VS Code interface with the 'index.html' file open in the editor. The code displays a basic HTML structure with a table:

```
<table>
  <thead>
    <tr>
      <th>CÓDIGO</th>
      <th>NOMBRE</th>
      <th>IMPORTE</th>
      <th>STOCK</th>
      <th>ELIMINAR</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td></td>
      <td></td>
      <td></td>
      <td></td>
      <td class="btn-eliminar" title="Eliminar producto">X</td>
    </tr>
  </tbody>
</table>
```

Fuente: captura de pantalla de VS Code.

**Index.html** tiene la estructura del formulario que trabajamos en el tema anterior, con el agregado de una tabla HTML vacía. Esta tiene dos partes: **<thead>** corresponde al encabezado de la tabla, y **<tbody>** como el cuerpo o contenido de la tabla.

Este último apartado está vacío y se completará con datos que se cargan a través del formulario HTML. Estos datos corresponden a un producto con ciertas características, se almacenarán primero en el **array productos**, y luego en LocalStorage, para poder mantener la persistencia de la información. LocalStorage nos servirá de base de datos local, ya que almacena la estructura del **array** de objetos correspondiente a los productos en sí que cargaremos.

Figura 32: Array productos y función guardarProducto()

The screenshot shows the VS Code interface with the 'formulario.js' file open in the editor. The code defines a function **guardarProducto()** that creates a new product object and adds it to an array:

```
const btnGuardar = document.querySelector('button#btnGuardar')
const inputCodigo = document.querySelector("input#inputCodigo")
const inputNombre = document.querySelector("input#inputNombre")
const inputImporte = document.querySelector("input#inputImporte")
const inputStock = document.querySelector("input#inputStock")

const productos = []

const obtenerCodigo = ()=> parseInt(Math.random() * 10_000)

function guardarProducto() {
  const nuevoProducto = { id: 0, nombre: '', importe: 0.00, stock: 0 }
}
```

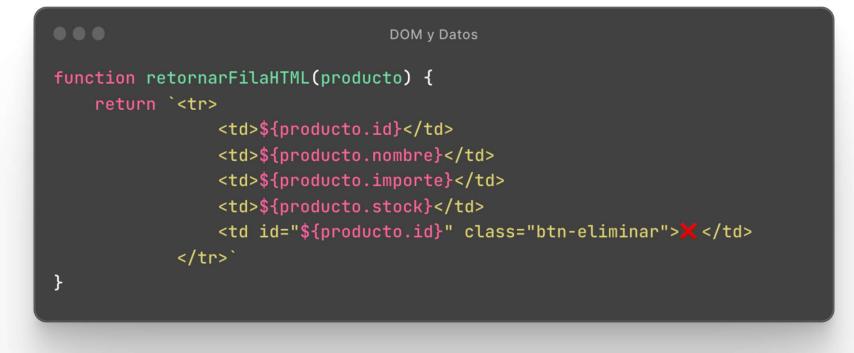
Fuente: captura de pantalla de VS Code.

El archivo JS posee un **array productos** vacío y una función llamada **guardarProducto()**. Esta posee un objeto literal que se completará según los datos cargados en el objeto literal modelo, que se guarda en el **array productos**. Finalmente, recorreremos el **array productos** y cargaremos cada producto en la tabla HTML.

## Estructura de tabla dinámica

Creamos una función con retorno, que contendrá la estructura de la fila de la tabla HTML. Esta función recibirá como parámetro un objeto literal. Este contendrá toda la información de cada producto que conforma el array. Agregamos, entonces, los *template literals* para dejar preparada esta función al momento de utilizarla.

Figura 33: Función con retorno



```
DOM y Datos

function retornarFilaHTML(producto) {
    return `<tr>
        <td>${producto.id}</td>
        <td>${producto.nombre}</td>
        <td>${producto.importe}</td>
        <td>${producto.stock}</td>
        <td id="${producto.id}" class="btn-eliminar"><span>X</span></td>
    </tr>`
}
```

Fuente: elaboración propia.

En este ejemplo, repetimos el uso de la propiedad ID, que será de utilidad cuando debamos definir los eventos para cada botón dinámico que se generará.

### Dar de alta un producto

Completemos la función **guardarProducto()** con la definición de la estructura del objeto literal modelo, que toma los valores cargados de cada campo del formulario HTML. El valor ID del mismo se genera de forma automática mediante la función **obtenerCodigo()**.

Figura 34: Funciones **guardarProducto()** y **obtenerCodigo()**



```
DOM y Datos

function guardarProducto() {
    const nuevoProducto = { id: obtenerCodigo(),
        nombre: inputNombre.value,
        importe: parseFloat(inputImporte.value),
        stock: parseInt(inputStock.value)
    }
    productos.push(nuevoProducto)
}

function obtenerCodigo() {
    let id = '';
    do {
        id = Math.floor(Math.random() * 1000000);
    } while (productos.find(p => p.id === id));
    return id;
}
```

Fuente: elaboración propia.

Con el objeto literal configurado, nos queda agregarlo al array **productos**. Por último, almacenamos este array en **LocalStorage**. Tengamos presente que, al ser un array de objetos, debemos convertirlo a través del objeto JSON y el método **stringify()**.

Figura 35: Convertir el array de objetos a través de JSON y **stringify()**

```
DOM y Datos

function almacenarProductosEnLS() {
    if (productos.length > 0) {
        localStorage.setItem("Productos", JSON.stringify(productos))
    }
}
```

Fuente: elaboración propia.

Creada esta última función, se la agrega al final de la función **guardarProducto()**. Por cada producto que damos de alta, guardamos el *array* completo en LocalStorage. Luego creamos la función **recuperarProductosDeLS()**, para cargar nuestro *array* apenas se cargue o recargue el documento HTML. Esta función retornará el *array* recuperado desde LocalStorage.

**Figura 36: Función recuperarProductosDeLS()**

```
DOM y Datos

function recuperarProductosDeLS() {
    return JSON.parse(localStorage.getItem("Productos"))
}
```

Fuente: elaboración propia.

Finalmente, creamos la función **cargarProductos()**. Esta recorrerá el *array* **productos** y cargará todo elemento que este posea en la tabla HTML. Para poder cargar los productos en la tabla, y crear cada fila de manera dinámica, debemos enlazarnos con el tag **<tbody>** para poder “escribir” el contenido HTML de cada fila de la tabla.

**Figura 37: Función cargarProductos()**

```
DOM y Datos

const tbody = document.querySelector("tbody")

function cargarProductos() {
    if (productos.length > 0) {
        tbody.innerHTML = ''
        productos.forEach((producto) => tbody.innerHTML += retornarFilaHTML(producto))
    }
}
```

Fuente: elaboración propia.

Esta última función también se agrega al final del bloque de código de la función **guardarProducto()**. De esta forma, armamos el objeto literal producto, lo agregamos al *array* **productos**, guardamos el *array* completo en LocalStorage y lo iteramos para cargar todo su contenido en la tabla HTML.

## Integración de DOM, datos y eventos

Agregamos ahora el evento **click** en el botón guardar del formulario HTML. Ante la ejecución de este evento, se recurre a la función **guardarProducto()**.

**Figura 38: Evento click**

```
DOM y Datos  
btnGuardar.addEventListener("click", guardarProducto)
```

Fuente: elaboración propia.

La función **recuperarProductosDeLS()** nos retorna el *array* productos que esté almacenado en LocalStorage. Asignamos esta función al establecer la constante **productos**. Agregamos el operador lógico OR para definir un *array* vacío, si es que no existe un *array* con elementos previamente guardado en LocalStorage.

**Figura 39: Función recuperarProductosDeLS()**

```
DOM y Datos  
const productos = recuperarProductosDeLS() || []
```

Fuente: elaboración propia.

### Almacenamiento y recuperación de datos

Solo nos queda referenciar la función **cargarProductos()** de modo que se ejecute apenas se cargue o recargue el proyecto. De esta forma, siempre veremos los productos que estén almacenados en LocalStorage, en la tabla HTML.

**Figura 40: Formulario HTML y tabla de productos funcionales**

Ventadefrutas.com

Ingresar nuevos productos

Código (autonumérico)  
Nombre del producto  
Importe (en \$)  
Stock (unidades)

CÓDIGO	NOMBRE	IMPORTE	STOCK	ELIMINAR
4016	MACBOOK AIR 13 - M2	749500	25	X
7311	NOTEBOOK 17 PULGADAS	125500	40	X
9495	TABLET IPAD PRO 10.1	215300	65	X

Fuente: elaboración propia.

El formulario HTML y la tabla de productos serán totalmente funcionales.

**Figura 41: Información almacenada en LocalStorage**

La captura de pantalla muestra una interfaz de desarrollo web con un sidebar izquierdo que incluye secciones para 'Aplicación' (Manifiesto, Service Workers, Almacenamiento) y 'Almacenamiento' (Almacenamiento local, http://127.0.0.1:5500/). En el panel central, se visualiza una tabla titulada 'Clave' y 'Valor'. La clave 'Productos' tiene un valor que es un array JSON: `[{"id":4016,"nombre":"MACBOOK AIR 13 - M2","importe":749500,"stock":25}, {"id":7311,"nombre":"NOTEBOOK 17 PULGADAS","importe":125500,"stock":40}, {"id":9495,"nombre":"TABLET IPAD PRO 10.1","importe":215300,"stock":65}]`. Una sección desplegable muestra los tres elementos individuales del array.

Fuente: elaboración propia.

La información almacenada en LocalStorage nos permite mantener los datos que se almacenan e incrementar los mismos de acuerdo con nuestra necesidad.

### Gestión de datos dinámicos

Para cerrar este ejemplo, se agrega de manera masiva el evento clic a cada uno de los botones “eliminar” creados en la tabla. Se crea una colección llamada **botonesEliminar**, a través de **querySelectorAll**. Recorremos esta colección y le asignamos a cada botón el evento clic. Este se ocupará de buscar el producto por el ID almacenado en el atributo ID de esta celda HTML. Una vez que se ubica el producto por su ID en el array **productos**, lo quitamos de la mano del método **splice()**.

**Figura 42: Colección botonesEliminar**

El código muestra una función que recorre todos los botones con la clase 'btn-eliminar'. Para cada botón, se agrega un oyente de clic que obtiene el ID del botón, busca el producto correspondiente en el array 'productos' y lo elimina usando el método 'splice'. Luego, se llama a las funciones 'almacenarProductosEnLS()' y 'cargarProductos()' para actualizar la lista.

```
const botonesEliminar = document.querySelectorAll("td.btn-eliminar")
for (let botonEliminar of botonesEliminar) {
    botonEliminar.addEventListener("click", ()=> {
        id = productos.findIndex(producto => producto.id ===
        parseInt(botonEliminar.id))
        if (id != -1) {
            productos.splice(id, 1)
            almacenarProductosEnLS()
            cargarProductos()
        }
    })
}
```

Fuente: elaboración propia.

Por último, se recurre a la función **almacenarProductosEnLS()** para actualizar el array, y a **cargarProductos()** para actualizar el contenido de la tabla.

### Reutilizar los eventos masivos

Este último bloque de código requiere encerrarse en una función. A esta se la llama directamente desde la función **cargarProductos()**. Esta última función se convoca por cada nuevo producto agregado y por cada producto que se elimine. Por lo tanto, necesitamos redefinir el evento clic de cada botón de la tabla, por cada vez que se refresquen los datos.

Para cerrar la lógica de este ejemplo práctico, creamos la función **activarClickEnBotonesEliminar()** y copiamos en ella todo el bloque de código anterior.

#### Actividad 3

En caso de no haber realizado la actividad de esta unidad, te invitamos a seguirla de acuerdo con cada bloque de código exemplificado, sobre el proyecto que se indica descargar al inicio del tema 3.

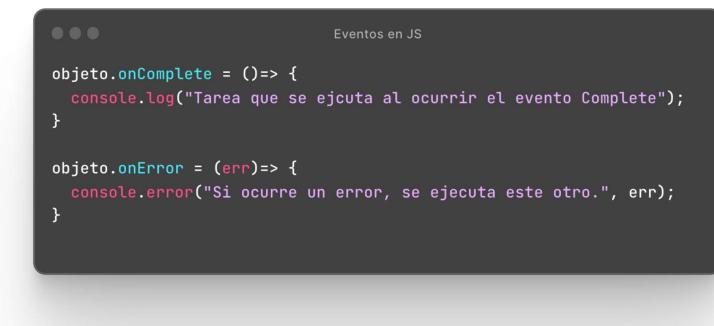
Este proyecto es 100 % funcional y permite aplicar los conceptos estudiados hasta el momento, relativos a los fundamentos de JavaScript, en un nivel avanzado.

Para conocer la resolución del caso, descargá el archivo al final de la lectura.

#### Tema 4: Promesas JavaScript

Una promesa JavaScript es un objeto que representa el resultado eventual de una operación asíncrona. Permite manejar flujos de código más legibles y evitar el anidamiento excesivo de callbacks. Este concepto fue clave para comenzar a evolucionar los eventos en JS, mientras que se aporta un valor agregado más importante dentro del mundo de las clases y objetos JS. El manejo de eventos en JS era inicialmente la forma más efectiva de esperar indefinidamente a que algo suceda, para luego reaccionar. Ese “algo” puede demorarse X cantidad de tiempo, que no podemos calcular, por ello debemos controlar la tarea siguiente mediante un evento que sí pueda esperar.

Figura 43: Eventos en JS



```
Eventos en JS

objeto.onComplete = ()=> {
    console.log("Tarea que se ejecuta al ocurrir el evento Complete");
}

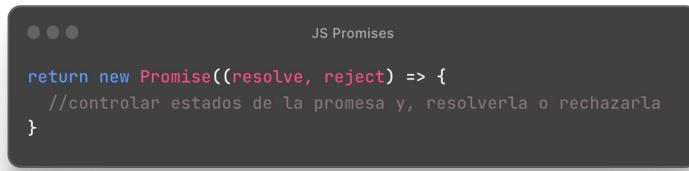
objeto.onError = (err)=> {
    console.error("Si ocurre un error, se ejecuta este otro.", err);
}
```

Fuente: elaboración propia.

#### Estados de una promesa

Las promesas se crean con la instancia de la clase *promise*, que acepta una función *callback* ejecutada asincrónicamente. Esta función, a su vez, acepta dos parámetros: **resolve** y **reject**. Veamos, a continuación, un ejemplo básico del uso de promesas a partir de la clase *promise*, para comprender su estructura básica.

Figura 44: Promesa JS. Clase promise



```
JS Promises

return new Promise((resolve, reject) => {
    //controlar estados de la promesa y, resolverla o rechazarla
})
```

Fuente: elaboración propia.

#### Promesa resuelta

Cuando una promesa se resuelve, se ejecuta el objeto **resolve**. En este debemos definir qué código se deberá ejecutar cuando sea llamado.

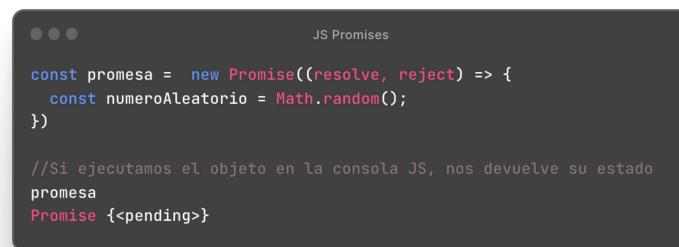
#### Promesa rechazada

Si por algún motivo, la promesa no puede llegar a realizar determinada operación, esto hará que la promesa sea rechazada, y con ello se ejecuta el objeto **reject**. Aquí definimos un código que controle el proceso de un rechazo, o hasta definir un reintento de operación.

## Estados de una promesa

Las promesas pueden disponer de tres estados diferentes (*pending*, *fulfilled*, *rejected*). Cuando nacen, asumen de forma predeterminada el primero de los estados: *pending*. Este proceso es manejado internamente, y cada estado se relaciona directamente con la ejecución de la promesa en sí.

Figura 45: Promesa JS. Estado *pending*



```
JS Promises

const promesa = new Promise((resolve, reject) => {
  const numeroAleatorio = Math.random();
})

//Si ejecutamos el objeto en la consola JS, nos devuelve su estado
promesa
Promise {<pending>}
```

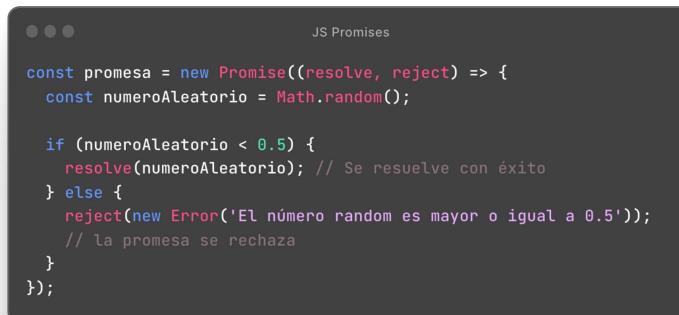
Fuente: elaboración propia.

Una vez que una promesa JS se haya resuelto, o rechazado, cambiará su estado inicial por el estado apropiado de acuerdo a su resolución o rechazo:

- ***fulfilled***,
- ***rejected***.

Veamos un ejemplo, a continuación, para entender cómo definir una promesa de acuerdo con nuestra necesidad. Este ejemplo genera un número aleatorio con **Math.random()**. En el caso de que este número aleatorio sea menor a **0.5**, la promesa se resuelve satisfactoriamente y retorna el número en cuestión.

Figura 46: Definir una promesa



```
JS Promises

const promesa = new Promise((resolve, reject) => {
  const numeroAleatorio = Math.random();

  if (numeroAleatorio < 0.5) {
    resolve(numeroAleatorio); // Se resuelve con éxito
  } else {
    reject(new Error('El número random es mayor o igual a 0.5'));
    // la promesa se rechaza
  }
});
```

Fuente: elaboración propia.

En caso de que el número retornado por **Math.random()** sea menor a 0.5, la promesa será rechazada y arrojará un error a la consola JS.

Tanto **resolve** como **reject** son objetos integrados a la promesa. Utilizan un retorno implícito del resultado que cada uno de ellos maneje. Si se resuelve la promesa y se ejecuta el objeto **Resolve()**, el estado de la promesa cambiará automáticamente a *fulfilled*.

Si la promesa falla y se ejecuta el objeto **Reject()**, el estado de la promesa cambiará automáticamente a **rejected**.

### Métodos de control

Toda promesa cuenta con métodos de control que permiten estructurar el código para ejecutarse de manera controlada. De esta manera se maneja de forma efectiva la respuesta de una promesa.

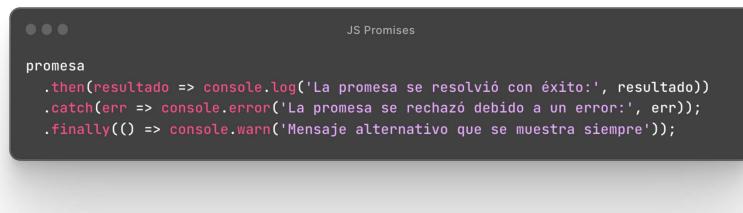
Tabla 7: Métodos de control

Estado	Descripción
.then(result)	Este método nos permite controlar cada tarea que se ejecuta luego de la respuesta de la promesa. Podemos encadenar tantos <b>.then()</b> como necesitemos. El parámetro <b>result</b> corresponde al valor o resultado entregado por la promesa, mediante el objeto <b>resolve()</b> .
.catch(err)	Cualquier error que surja, o ante el rechazo de la promesa, el error será controlado por el método <b>.catch()</b> . El error resultante pasa con el parámetro <b>err</b> , o <b>error</b> .
.finally()	Este método es opcional y se ejecutará siempre, independientemente de cuál haya sido el estado resultante de la promesa <b>resolve()</b> o <b>reject()</b> .

Fuente: elaboración propia.

Veamos a continuación un ejemplo de estos métodos aplicados a la promesa anterior.

Figura 47: Ejemplo de métodos aplicados



```
JS Promises

promesa
  .then(resultado => console.log('La promesa se resolvió con éxito:', resultado))
  .catch(err => console.error('La promesa se rechazó debido a un error:', err));
  .finally(() => console.warn('Mensaje alternativo que se muestra siempre'));
```

Fuente: elaboración propia.

Debemos contemplar también que, cuando una promesa se resuelve o rechaza y cambia de estado, este será el definitivo y no podremos volver atrás.

### Ejemplos aplicados

Las promesas son herramientas que podemos utilizar para crear objetos asíncronos basados en nuestras necesidades específicas, pero también contamos con la integración de promesas en determinadas herramientas cotidianas de JS.

Este concepto abstracto se aplica más adelante cuando se trabaja con la petición de datos a servidores. También se pueden utilizar promesas integradas en librerías JS, por ejemplo: la librería de Sweet alert o Alertify, entre otras.

#### Actividad 4

Recuperar el proyecto trabajado en las dos microactividades anteriores. Al final del mismo, transcribir el ejemplo de promesas abordado en este tema.

Modificar el proyecto para que, al comparar la constante **numeroAleatorio**, esta se resuelva cuando su valor sea mayor a 0.5.

Cuando se resuelve la promesa, retornar el array **productos** creado y poblado previamente, de acuerdo con lo planteado en la **microactividad 2**.

Si la promesa es rechazada, retornar un error que diga “El número obtenido es menor a 0.5”.

Debajo, convocar a la promesa y controlar su respuesta con el método **then()**, donde posiblemente se reciba como parámetro el array **productos**. Si este array llega, definir dentro del método de control **console.table(productos)** para que estos se visualicen en la consola JS. Agregar también el control de errores con el método **catch()**, que debe recibir como parámetro el error generado en el apartado **reject()**.

Para conocer la resolución del caso, descargá el archivo al final de la lectura.

## Unidad 2: AJAX y Fetch

En este módulo de estudio se estudian conceptos esenciales para comprender y trabajar con tecnologías web modernas de la mano del paradigma AJAX, que revolucionó la forma en que las aplicaciones web interactúan con los servidores.

Continuamos con la comprensión de las peticiones HTTP, los métodos utilizados para comunicarse con los servidores, y cómo podemos aprovecharlos para enviar y recibir datos. Además, examinaremos los códigos de estado y las respuestas del servidor, que proporcionan información valiosa sobre el resultado de las solicitudes.

El módulo de estudio se cierra con la descripción de los diferentes formatos de transporte de datos, como JSON y XML, que permiten intercambiar información estructurada entre el cliente y el servidor. Con los conocimientos adquiridos a lo largo de este documento, se podrán construir aplicaciones web interactivas y eficientes.

### Tema 1: El modelo cliente-servidor

Repasemos algunos conceptos: el protocolo HTTP (*hypertext transfer protocol*), utilizado para intercambiar información a través de internet, fue creado para transferir datos según el modelo cliente-servidor. Este modelo envía una petición al servidor, que recibe una respuesta asociada. El cliente es quien inicia la comunicación, y el servidor es quien se ocupa de responder. Todo este proceso se realiza a través de la tecnología WebSocket.

El modelo de intercambio de datos que aquí se utiliza es el de un protocolo sin estado. Este protocolo en sí no regula qué sucede dentro del servidor ni del lado del cliente. Solo se ocupa de establecer la comunicación entre las partes, y de regular la entrada y salida de datos entre ambos agentes, a través del intercambio de mensajes.

#### Qué es el paradigma AJAX

El mismo proceso que se involucra en el modelo cliente-servidor, utilizado en redes de datos corporativas y llevado como base para el funcionamiento de la red internet, fue el puntapié inicial para que el paradigma AJAX cobrara vida.

Hacia los años 2002 y 2003, Microsoft ideó una propuesta original, denominada AJAX, basada en el funcionamiento de las peticiones HTTP. El objetivo era que cualquier tipo de tecnología *frontend* y *backend* pudiese intercambiar datos con cualquier otra tecnología opuesta.

Esta tecnología (*asynchronous JavaScript and XML*) propuso utilizar una URL convencional para pedir información (datos) a un servidor (cuálquiera), y que dicha información sea transferida mediante el mismo protocolo HTTP a la computadora cliente, en un formato estandarizado.

AJAX logró no solo unificar información entre diferentes tecnologías de cliente y servidor, sino también aprovechar el

intercambio de documentos web convencionales (imágenes, HTML, CSS, XML, JavaScript, entre otros), sin tener que recargar (*refresh - reload*) todo el documento HTML.

### Peticiones HTTP

Para el funcionamiento de AJAX, se eligió continuar con el uso del modelo de peticiones HTTP. Estas proponen realizar pedidos al servidor, a través de una URL, y usar un cliente como gestor de dicha información.

**Figura 48: Peticiones HTTP**



Fuente: elaboración propia.

La URL oficia de medio de transporte de la información que se busca obtener. En el ejemplo anterior, vemos que se le pide información de **/usuarios/** a un servidor. El último parámetro indica que se busca obtener la información de todos los usuarios **/all/**.

De esta forma se logró utilizar la URL como la forma más oportuna para obtener información específica desde el servidor. Este mecanismo es conocido como **endpoint**.

**Figura 49: Endpoint**

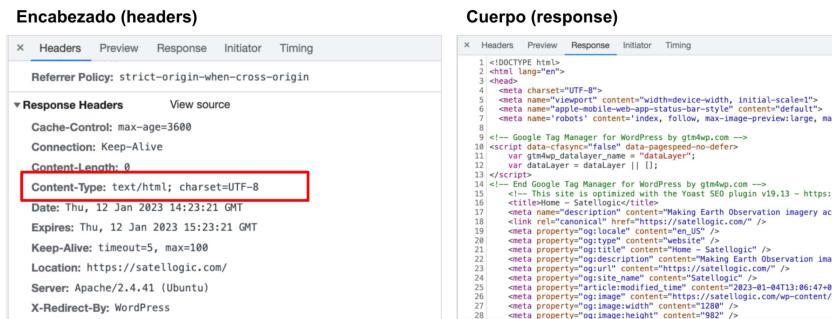


Fuente: elaboración propia.

### Encabezado y cuerpo

La información descargada vía *stream* posee un encabezado y un cuerpo. En el encabezado (*header*) viaja información diversa, entre ella, el estado de la petición. En el cuerpo (*body*) viaja la información referente al recurso que descargamos.

**Figura 50: Encabezado y cuerpo**



The screenshot shows the browser's developer tools Network tab. A specific response is selected, and its Headers section is visible. The Content-Type header is highlighted with a red box. Other headers shown include Referrer Policy, Cache-Control, Connection, Content-Length, Date, Expires, Keep-Alive, Location, Server, and X-Redirect-By.

```

Content-Type: text/html; charset=UTF-8
Date: Thu, 12 Jan 2023 14:23:21 GMT
Expires: Thu, 12 Jan 2023 15:23:21 GMT
Keep-Alive: timeout=5, max=100
Location: https://satellogic.com/
Server: Apache/2.4.41 (Ubuntu)
X-Redirect-By: WordPress

```

Fuente: elaboración propia.

Existen formas nativas de preguntar, mediante JavaScript, si una web API está o no disponible en el navegador o en el sistema operativo donde corre la aplicación web. De esta forma, evitaremos que nuestra aplicación web falle, y hasta podemos buscar una alternativa de funcionamiento si la API web no está disponible.

### Códigos de estado y respuesta

Con cada petición HTTP de respuesta viaja un código de estado, dentro del encabezado. Estos códigos son, básicamente, datos relacionados con la respuesta del servidor a nuestra petición. Los códigos son diferentes, y varían de acuerdo al tipo de petición. Se ocupan de confirmar si todo ha ido bien o si surgió algún error durante la descarga del recurso. Veamos, a continuación, un resumen de los códigos de estado, de acuerdo con el tipo de numeración y el significado de esta.

**Tabla 8: Códigos de estado y respuesta**

Código	Descripción
<b>100</b>	códigos informativos
<b>200</b>	códigos de éxito en la petición
<b>300</b>	códigos de redireccionamiento
<b>400</b>	códigos de error en el cliente
<b>500</b>	códigos de error en el servidor

Fuente: elaboración propia.

Consulta el siguiente artículo si quieres ampliar conocimientos sobre los códigos de estado:

**Ofiwe, M. (2021). Códigos de estado HTTP: una guía sin tecnicismos. Semrush Blog. <https://es.semrush.com/blog/codigos-de-estado-http/>.**

### Códigos de estado exitosos

El rango de códigos de estado basados en 200 (201, 202, etc.) indica que la respuesta a la petición se ha realizado de manera exitosa.

**Figura 51: Rango de códigos de estado de respuesta exitosa**

The screenshot shows a network request details panel. At the top, there are tabs: Headers, Payload, Preview, Response, Initiator, Timing, and Cookies. The Headers tab is selected. Below the tabs, under the 'General' section, the Request URL is https://www.google.com/recaptcha/api2/anchor?ar=1&k=6Ld9DPEbAAAAANazJS05FSKTviGsisMjfVYFjyV&co=aHR0cHM6Ly9zYXRlbGxvZ2ljLmNvbTo0NDM.&hl=es-419&v=5qcenVbrh0y8zihcc2aH0WD4&size=invisible&cb=8fo0ghqk3ont. The Request Method is GET. A red box highlights the Status Code: 200. Other details shown include Remote Address: [2800:3f0:4002:812::2004]:443 and Referrer Policy: strict-origin-when-cross-origin.

Fuente: elaboración propia.

### Códigos de estado erróneos

El rango de códigos de estado basados en 400 (401, 402, etc.) indica que ocurrió un error al intentar obtener un recurso remoto. El más conocido de ellos: el error 404.

Figura 52: Rango de códigos de estado con respuesta de error

The screenshot shows a network request details panel. The Headers tab is selected. Under the 'General' section, the Request URL is https://satellogic.com/images/pepe-argento.jpg, the Request Method is GET, and the Status Code is 404 Not Found (highlighted with a red box). Other details include Remote Address: 54.73.143.100:443 and Referrer Policy: strict-origin-when-cross-origin. Below the General section, there is a 'Response Headers' section with a single entry: Cache-Control: no-cache, must-revalidate, max-age=0.

Fuente: elaboración propia.

### Formatos de transporte de datos

Cuando hablamos de intercambiar información (datos) con un servidor, existe un formato de transporte de datos estandarizado. Cuando nació la tecnología AJAX, se optó por el formato XML, popular en aquel momento. A esto corresponde la X de este paradigma.

Algunos años después, nace el formato JSON, inspirado en la estructura de objetos y propiedades que maneja el lenguaje JS. Veamos, a continuación, un gráfico que muestra el mismo tipo de datos de respuesta por parte de un servidor, representada por XML y por JSON.

Figura 53: Estructura de datos: XML y JSON

## estructura de datos XML

```
Status: 200 OK Size: 1.66 KB Time: 281 ms
Response Headers[16] Cookies Results Docs (-) [?]
1  <?xml version="1.0" encoding="UTF-8"?>
2  <user>
3    <result>
4      <gender>female</gender>
5      <name>
6        <titleMiss>title</titleMiss>
7        <firstName>Troyana</firstName>
8        <lastName>Burko</lastName>
9      </name>
10     <location>
11       <city>Izmir</city>
12       <number>6097</number>
13       <name>B&B Lintya</name>
14     </street>
15     <country>Ukraine</country>
16     <stateOrProvince>Lviv</stateOrProvince>
17     <postcode>41276</postcode>
18     <coordinates>
19       <latitude>47.5584</latitude>
20       <longitude>21.0579</longitude>
21     </coordinates>
22     <descriptions>
23       <offset>+0:00</offset>
24       <description>Beijing, Perth, Singapore, Hong Kong</description>
25     </descriptions>
26   </location>
27   <email>troyana.burko@example.com</email>
28 <login>
```

## estructura de datos JSON

```
Status: 200 OK Size: 1.15 KB Time: 684 ms
Response Headers[16] Cookies Results Docs (-) [?]
1  {
2    "results": [
3      {
4        "gender": "male",
5        "name": {
6          "title": "Mr",
7          "first": "Pascual",
8          "last": "Armas"
9        },
10       "location": {
11         "street": {
12           "number": 9459,
13           "name": "Creador Vera"
14         },
15         "city": "Tlalnepantla",
16         "state": "Veracruz",
17         "country": "Mexico",
18         "postcode": 93596,
19         "coordinates": {
20           "latitude": "23.2536",
21           "longitude": "-51.3049"
22         },
23         "timezones": {
24           "offset": "-05:00",
25           "description": "Ekaterinburg, Islamabad, Karachi, Tashkent"
26         }
27       },
28     }
29   },
```

Fuente: elaboración propia.

El formato XML se asemeja mucho a la estructura HTML, y el formato JSON se asimila al manejo de objetos y propiedades de JavaScript.

Cuando manejamos grandes volúmenes de datos de intercambio entre un servidor y un cliente, el formato JSON es el más apropiado, dado que utilizará mucha menos estructura base para representar los datos. Por lo tanto, el tamaño de respuesta del servidor será significativamente menor que enviar los mismos datos en formato XML.

### Actividad 5

- 1) Cuando descargamos una petición realizada al servidor, la información vía stream retorna solamente un cuerpo. El encabezado queda almacenado en el servidor como validación de la información que retornó.

Verdadero

Falso

### Justificación

- 2) Los códigos de estado 200 indican que el estado de la solicitud es válida o correcta, mientras que los códigos de estado 400 indican una solicitud errónea.

Verdadero

Falso

### Justificación

## Tema 2: Aplicaciones de servidor

Estas aplicaciones son programas que se ejecutan en el lado del servidor y se encargan de procesar las solicitudes que envían los usuarios a través de internet. Son responsables de manejar la lógica y la funcionalidad de una página web o una aplicación (nativa o móvil), almacenar y acceder a la información en una base de datos, y enviar las respuestas adecuadas a los usuarios.

### Qué es una aplicación de *backend*

Una aplicación de *backend* es, básicamente, una aplicación de servidor que se encarga de procesar las solicitudes provenientes de los clientes, realizar operaciones en la base de datos y enviar las respuestas correspondientes. Es responsable de la lógica de negocio, la seguridad y el almacenamiento de datos, de esta forma permite la interacción entre el cliente y el servidor en una aplicación web.

A diferencia de las aplicaciones web frontend, las aplicaciones de servidor no suelen tener una interfaz gráfica. Funcionan como un proceso cargado en memoria, que escucha peticiones, dialoga con servicios o aplicaciones (en el servidor o remotas), y

devuelve respuestas al cliente que realizó la petición.

Figura 54: Modelo de aplicaciones frontend y backend



Fuente: elaboración propia.

En el gráfico anterior se muestra una representación completa del modelo cliente-servidor. A la izquierda se ven dispositivos de todo tipo (computadoras, navegadores web, apps móviles, dispositivos electrónicos IoT, electrodomésticos modernos). Estos utilizan internet (centro) como medio de transporte para pedir o almacenar información con una aplicación de servidor (derecha).

Del lado del servidor no solamente se encuentra la aplicación de *backend* en sí, sino también otros actores importantes, como los mecanismos de seguridad, las bases de datos de diferentes tipos, los recursos de almacenamiento, etcétera.

### Tecnologías

Actualmente contamos con una importante variedad de tecnologías que permiten construir aplicaciones de servidor. Estas son indistintas porque, gracias al paradigma AJAX, todas utilizan un lenguaje común para poder intercambiar datos entre el servidor y el o los clientes que soliciten dicha información.

A continuación, se presenta un listado de tecnologías en formato combo, que suelen utilizarse hoy del lado del servidor.

Tabla 9: Stacks de desarrollo de aplicaciones *backend*

Tecnología	Compuesta por
LAMP	<ul style="list-style-type: none"><li>Lenguaje de programación: PHP.</li><li>Servidor web: Apache.</li><li>Motor de base de datos: MySQL.</li><li>Sistema operativo: Linux.</li></ul>
MEN	<ul style="list-style-type: none"><li>Lenguaje de programación: JavaScript (Node.js).</li><li>Servidor web: Express.js.</li><li>Motor de base de datos: MongoDB.</li></ul>
Django	<ul style="list-style-type: none"><li>Lenguaje de programación: Python.</li><li>Framework de <b>backend</b>: Django.</li><li>Motor de base de datos: PostgreSQL.</li></ul>
Ruby on rails	<ul style="list-style-type: none"><li>Lenguaje de programación: Ruby.</li><li>Framework de <b>backend</b>: Ruby on rails.</li></ul>

	<ul style="list-style-type: none"> <li>• Motor de base de datos: SQLite, PostgreSQL o MySQL.</li> </ul>
<b>ASP.NET</b>	<ul style="list-style-type: none"> <li>• Lenguaje de programación: C# (ASP.NET).</li> <li>• <i>Framework de backend</i>: ASP.NET.</li> <li>• Motor de base de datos: SQL Server.</li> </ul>
<b>JAVA</b>	<ul style="list-style-type: none"> <li>• Lenguaje de programación: Java.</li> <li>• <i>Framework de backend</i>: Spring (Spring Boot, Spring MVC).</li> <li>• Servidor web: Apache Tomcat, Jetty.</li> <li>• Motor de base de datos: Oracle (usando Hibernate o Spring Data).</li> </ul>

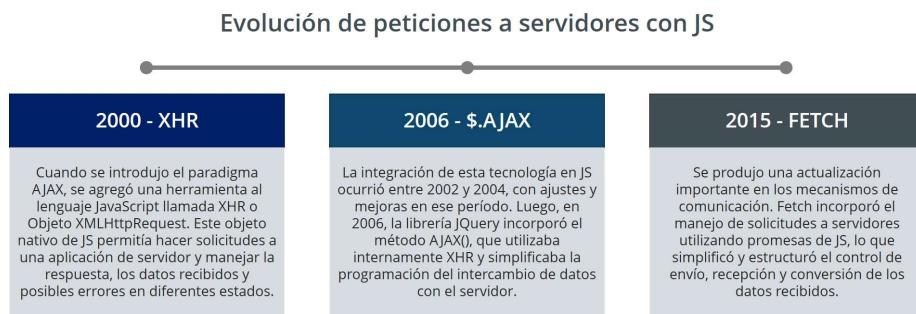
Fuente: elaboración propia.

En el ecosistema tecnológico actual, es corriente desarrollar soluciones de *software web* que trabajen contra cualquiera de las tecnologías anteriormente mencionadas. Incluso, existen muchos proyectos, ya constituidos y con años de funcionamiento, que por alguna cuestión técnica o estratégica deciden cambiar su plataforma de *backend* desde una tecnología a otra.

Gracias a AJAX, ese cambio no provocará ningún impacto a nivel funcional con aplicaciones *frontend*, dado que el idioma de intercambio de datos será seguramente JSON y, posiblemente, XML.

### Evolución de JS con servidores

**Figura 55: Evolución de peticiones a servidores con JS**



Fuente: elaboración propia.

Cuando nace el paradigma AJAX, este integra una herramienta al lenguaje JavaScript que anteriormente no existía. La misma se conoce como **XHR** u Objeto **XMLHttpRequest**. Este objeto nativo en el Core de JS permitía realizar peticiones a una aplicación de servidor, y manejar, a través de diferentes estados, la respuesta del servidor, la recepción de datos o cualquier posible error que ocurra en el medio.

Esta tecnología se integró a JS entre 2002 y 2004, y ajustó algunas funcionalidades en esta etapa. Luego, en 2006, la librería JQuery presentó, como parte de sus funcionalidades, el método **AJAX()**. Este utilizaba XHR internamente, pero simplificaba en parte la forma de programar el intercambio de datos con un servidor.

Finalmente, en 2015 llega una actualización importante de estos mecanismos de comunicación de JS con los servidores, de la mano de la función *fetch*. Esta integra internamente el manejo de peticiones a servidores, a través de promesas JS. De esta forma, el control de envío, recepción, y conversión datos recibidos era mucho más simple y estructurado.

**Figura 56: Evolución de peticiones a servidores con JS**

### Evolución de peticiones a servidores con JS

XHR	\$.AJAX	FETCH
2002	2006	2015

Fuente: elaboración propia.

Fetch es actualmente la opción más efectiva y más simple para dialogar con servidores de *backend*. Más allá de esto, siempre es bueno conocer los mecanismos anteriores y probarlos, al menos una vez, para entender cómo trabajan.

En el ecosistema IT no solo se construyen aplicaciones web con tecnologías modernas. También aparece la necesidad de investigar el funcionamiento de *software legacy* para migrar su core a nivel tecnológico, lo cual hará que podamos llegar a encontrarnos con el uso de **XHR** o **\$.ajax()**.

### El concepto REST API

REST API (*representational state transfer application programming interface*) es un conjunto de principios y convenciones para diseñar y desarrollar servicios web que se comunican a través del protocolo HTTP.

Este sigue el estilo arquitectónico REST. Una API REST utiliza los métodos estándar de HTTP (*get, post, put y delete*) para crear, leer, actualizar y eliminar recursos en un sistema o de un servidor.

Las API REST son altamente escalables, independientes de plataforma, y permiten una comunicación eficiente entre el cliente y el servidor. Utilizan formatos de datos como JSON o XML para intercambiar información estructurada. Este concepto (API REST) unifica bajo un mismo paraguas a todos los temas estudiados en este módulo.

Toda aplicación *frontend* que diseñemos (web, móvil, nativa), requiere de un servidor y de un concepto como API REST para poder acceder o crear datos que le den contexto y lógica.

### Servidores gratuitos

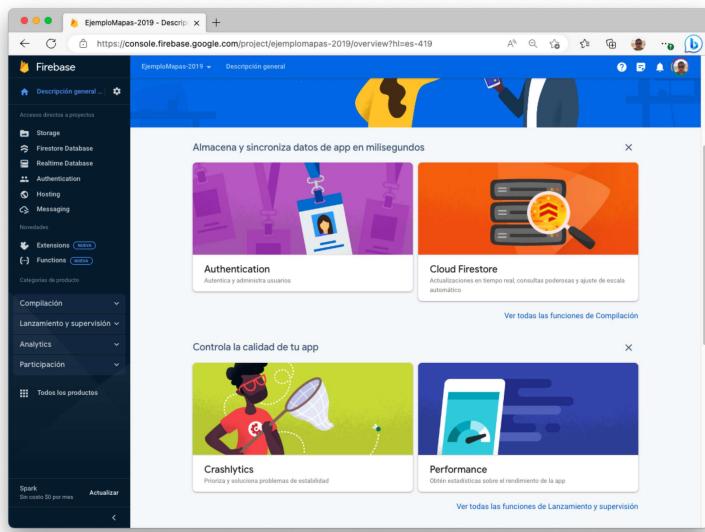
Para conseguir una experiencia completa en el trabajo contra aplicaciones de *backend*, tenemos la posibilidad de crearlas desde cero con la tecnología que más nos guste o que más conoczamos. También contamos con la posibilidad de utilizar servicios disponibles en internet, gratuitos o pagos, de acuerdo con el tipo de proyecto que necesitemos consolidar.

Existen una serie de servicios que permiten explorar estos terrenos con un esfuerzo mínimo o nulo. A continuación, se destacan las opciones más rápidas que podemos implementar.

### Firestore

Cualquiera que posea una cuenta de Gmail, tiene acceso a los servicios de Firestore. Esta es una plataforma de servicios *cloud*, de Google, con características mucho más simples que la que propone Google Cloud Platform.

**Figura 57: Firebase**



Fuente: elaboración propia.

Dentro de esta plataforma podemos crear muchos tipos de servicios, entre ellos, una base de datos con una estructura similar a NoSQL o datos JSON, y peticionar estos datos a través de módulos que brinda Firebase.

Su curva de preparación no es simple, pero tampoco tan compleja como otros servicios *cloud*. Es bueno tenerla en cuenta no solo para proyectos simples, sino también para los de complejidad media, dado que no solamente podemos almacenar y consultar datos, sino también podemos sumar servicios de autenticación, *testing* automatizado, *hosting*, *messaging*, entre otros.

### Actividad 6

Responder las siguientes preguntas, con base en la lectura de este último tema.

**1) La función Fetch() utiliza el mecanismo de petición de información a servidores y funciona internamente mediante el uso de:**

Evento *loaded*.

Evento *onsuccess*.

Promesas

#### Justificación

**2) ¿Cuáles son las principales características de las API REST?**

Altamente escalables.

Independientes de plataforma.

Utilizan formatos de datos JSON o XML.

A, b y c son correctas.

A, b y c son incorrectas.

#### Justificación

### Tema 3: Simular un backend

En el ecosistema de desarrollo de *software*, cuando un proyecto comienza a construirse en paralelo (*frontend* y *backend*), es muy factible que el desarrollo *frontend* sea el que primero avance. En este caso, llega mucho más rápido a necesitar de un set

de datos para aquellos módulos o páginas que requieren mostrar información dinámica. Para esto, existen varias técnicas que simulan un set de datos provenientes de un *backend*.

Veamos cómo sacar provecho de las estructuras de tipo *mockup*, hasta que la aplicación de *backend* esté disponible para ser integrada con nuestro *frontend*.

#### Archivos en formato JSON

Los archivos en formato JSON (*JavaScript object notation*) son una forma común de almacenar y compartir datos estructurados. Se trata de un formato ligero y legible, que se utiliza ampliamente en aplicaciones web y sistemas de intercambio de datos.

Estos archivos se componen de pares clave-valor, donde los datos se organizan en una estructura similar a un diccionario. Los valores pueden ser de diferentes tipos: cadenas de texto, números, booleanos, arrays o incluso otros objetos JSON anidados.

**Figura 58: Archivos en formato JSON**



```
{  
  "codigo": "P001",  
  "nombre": "Camiseta",  
  "precio": 19.99,  
  "stock": 100  
}
```

Fuente: elaboración propia.

El formato JSON es ampliamente utilizado en el desarrollo de aplicaciones, debido a su facilidad de lectura y escritura por parte de los humanos, así como su interoperabilidad con diferentes lenguajes de programación.

#### Diferencias con array de objetos

A simple vista, una estructura de datos JSON puede parecer similar a la estructura de *array* de objetos en JS, sin embargo, ambas son completamente diferentes.

JSON es un formato basado en texto y requiere que su estructura lo respete. A su vez, un *array* de objetos en JavaScript es una colección ordenada de objetos. Cada uno de estos objetos, además de representar datos en formato clave-valor, puede contener métodos internos.

Los datos en formato JSON no soportan ningún tipo de estructuras por fuera de los pares clave-valor. Y estas estructuras deben respetar el formato de comillas, tanto para las propiedades como también para todo dato que no sea numérico.

**Figura 59: Comparación entre formato JSON y formato array de objetos**

```

frutas.json
1 [ 
2   {
3     "id": 1,
4     "imagen": "🍌",
5     "nombre": "Bananas",
6     "importe": 220,
7     "stock": 50
8   },
9   {
10    "id": 2,
11    "imagen": "🍎",
12    "nombre": "Manzanas",
13    "importe": 270,
14    "stock": 50
15  },
16  {
17    "id": 3,
18    "imagen": "❓",
19  }
]

productos.js
1 const productos = [
2   {
3     id: 1,
4     imagen: "🍌",
5     nombre: "Bananas",
6     precio: 220,
7     stock: 50
8   },
9   {
10    id: 2,
11    imagen: "🍎",
12    nombre: "Manzanas",
13    precio: 200,
14    stock: 50
15  },
16  {
17    id: 3,
18    imagen: "❓",
19  }
]

```

Fuente: elaboración propia.

En el ejemplo anterior vemos una comparación de la misma información en un formato JSON (izquierda), y en un formato del tipo *array* de objetos (derecha).

También debemos tener claro que ningún archivo JSON soporta declaración de variables, constantes, ni nada que tenga que ver con JS. Tampoco soporta comentarios. Más allá de su similitud con una estructura JS, no tiene nada que ver uno con otro.

### Limitaciones de los archivos JSON

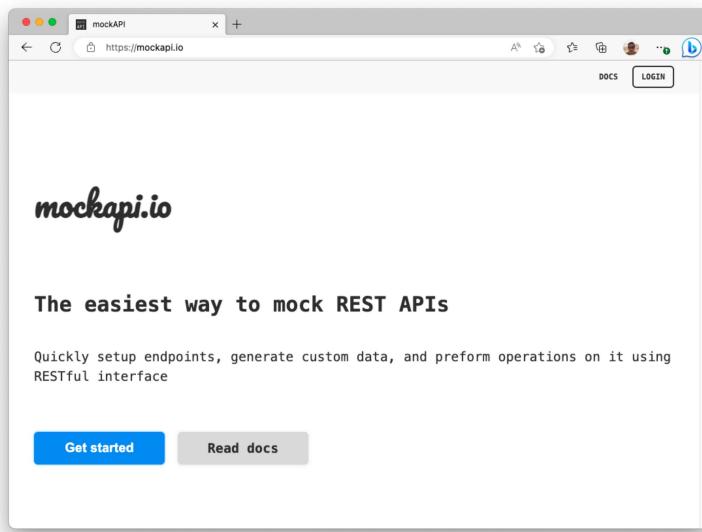
Los archivos JSON se utilizan, generalmente, para armar estructuras de configuración de parámetros, menús, o estructuras de datos orientadas a la construcción de *mockups*. Estos últimos permiten construir la parte gráfica de toda aplicación web, sin tener que depender de un *backend* para el acceso a datos.

Los archivos JSON nos ayudarán, entonces, a poder acelerar el trabajo *frontend*. Sin embargo, la información de estos podrá ser leída, pero no podrá ser escrita o modificada, al menos no de una forma fácil. Por lo tanto, es necesario tener siempre presente estas limitaciones y solo utilizar archivos JSON para *mockup* o simulación.

### Servicios cloud de *backend*

MockAPI es una plataforma que permite crear y simular API REST de manera rápida y sencilla. Posee un entorno donde definir *endpoints*, estructurar los datos de respuesta y simular diferentes escenarios. Es muy útil para desarrollo y pruebas de aplicaciones que requieren interactuar con una API, ya que otorga una visión real al ambiente de producción.

**Figura 60: MockAPI**



Fuente: elaboración propia.

MockAPI ofrece una capa gratuita, con limitaciones en cuanto a la cantidad de solicitudes y almacenamiento. Este servicio es práctico y suficiente para pruebas simples.

Nos registramos con nuestra cuenta de Google o Github, definimos una API y la estructura de datos que tendrá. Luego, se pega un *mockup* de datos en formato JSON para comenzar a utilizar MockAPI de manera remota.

#### Actividad 7

Acceder al siguiente sitio web: <https://codebeautify.org/jsontoxml/>.

Copiar el siguiente array de productos, directamente desde este documento.

```
[{"id": 1,"nombre": "Laptop","stock": 10,"precio": 1500,"categoria": "Computadoras"},  
 {"id": 2,"nombre": "Monitor","stock": 5,"precio": 300,"categoria": "Accesorios"},  
 {"id": 3,"nombre": "Teclado","stock": 20,"precio": 50,"categoria": "Accesorios"},  
 {"id": 4,"nombre": "Mouse","stock": 15,"precio": 20,"categoria": "Accesorios"},  
 {"id": 5,"nombre": "Impresora","stock": 8,"precio": 200,"categoria": "Periféricos"}]
```

Pegar el array de productos copiado, en la ventana lateral izquierda del sitio web.

Verificar cómo queda convertido este set de datos al formato XML.

Para conocer la resolución del caso, descargá el archivo al final de la lectura.

#### Tema 4: Fetch

Llegó el momento de utilizar la función *fetch* para poder acceder a datos. La misma nos brinda un mecanismo fácil y limpio para peticionar información, ya sea a través de archivos JSON locales o de endpoints locales o remotos.

Veamos cómo sacar provecho de esta función nativa de JavaScript, que será de gran utilidad para acceder a diferentes tipos de información, e importante para los desarrollos de aplicaciones web. Además, veremos cómo utilizarla para realizar las diferentes operaciones del tipo CRUD sobre una plataforma que las admita.

#### La función *fetch*

Podemos entender a la función **fetch()** como una función nativa de JS que permite hacer solicitudes de red desde el navegador web. La misma evolucionó el objeto **XHR** y el método **JQuery.AJAX()**, simplificó, en su máxima expresión, el funcionamiento de peticiones de datos a un servidor.

El funcionamiento estándar de **fetch()** se da a través del mecanismo de promesas. Peticiona datos y retorna la respuesta del servidor en el formato en el que envió los datos. Luego, con los métodos **.then()**, podemos transformar los tipos de datos de respuesta del servidor, cargarlos en un documento HTML, controlar cualquier posible error, etcétera.

**Figura 61: Función *fetch***



```
Fetch

fetch('http://example.com/movies.json')
  .then(response => response.json())
  .then(data => console.table(data));
```

Fuente: elaboración propia.

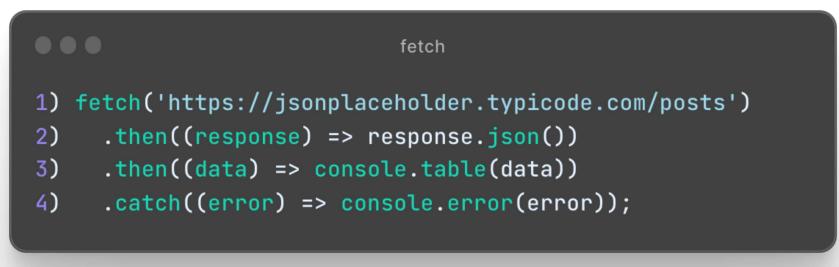
Su estructura es simple. Casi siempre que peticionamos datos en formato XML o JSON, a un servidor, la estructura que debemos definir para esta petición es casi una receta. Veamos un ejemplo, a continuación.

#### Manejo de *fetch* con promesas

JSONPlaceholder es un sitio web que posee un *mockup* de datos con el cual podemos interactuar, a través de diferentes mecanismos de petición, entre ellos **fetch()**. Para poder entender cómo se comporta *fetch()*, peticionaremos la URL <https://jsonplaceholder.typicode.com/posts>. Esta devolverá una serie de *posts* ficticios, como si se tratase de información a representar en un blog de noticias.

A continuación, se muestra un ejemplo de código para peticionar esta URL. Se analiza cada una de las líneas de código que componen el ejemplo, para entender qué sucede.

**Figura 62: Ejemplo de código para peticionar https://jsonplaceholder.typicode.com/posts**



```
fetch

1) fetch('https://jsonplaceholder.typicode.com/posts')
2)   .then((response) => response.json())
3)   .then((data) => console.table(data))
4)   .catch((error) => console.error(error));
```

Fuente: elaboración propia.

**Tabla 10: Referencia del manejo de peticiones *fetch()* mediante promesas**

Línea	Descripción
1)	Ejecutamos <b>fetch</b> , al pasar la URL como parámetro. Si el servidor está disponible, retornará los datos a nuestra petición. <i>Fetch</i> funciona con promesas, por lo tanto, si la promesa se cumple, esta función retorna el set de datos de forma implícita.
2)	El primer método <b>.then()</b> encadenado recibe un parámetro. Se suele representar este parámetro con la palabra <b>response</b> . En él se acarrea toda la respuesta del <i>endpoint</i> peticionado.

	El servidor responde siempre con datos en formato <i>string</i> , y sabemos de antemano que dichos datos son una estructura JSON. Entonces, podemos utilizar el método <code>.json()</code> nativo, integrado a la respuesta del servidor, para convertir dicha respuesta en un <i>array</i> de objetos JS. El método <code>.json()</code> realiza dicha conversión y retorna también de manera implícita el set de datos, transformado ya en un <i>array</i> de objetos JS.
3)	El segundo método <code>.then()</code> encadenado recibe como parámetro los datos ya transformados que retornó el primer método <code>then()</code> . Aquí ya podemos hacer uso de dichos datos, tal como si se tratase de un <i>array</i> de objetos nativos en JS. Podemos iterarlos para cargar la información en un documento HTML o, como vemos en nuestro ejemplo, representarlos en una tabla a través del objeto <code>console</code> .
4)	Si ocurre algún error desde la ejecución de <code>fetch()</code> , o en cualquier parte interna de los métodos <code>then()</code> encadenados, la lógica de la promesa hará que esta sea rechazada. En caso de que esto ocurra, podemos controlar cualquier posible error con el método <code>.catch()</code> . En este método pasamos como parámetro a <code>error</code> , que asume el valor del objeto global <code>error</code> . Podremos mostrar el error en cuestión, o directamente manipularlo y mostrar al usuario algún mensaje amigable.

Fuente: elaboración propia.

Esta receta es común para casi todos los usos básicos que daremos a las peticiones `fetch()`. Se puede probar esto en la consola JS del navegador.

Fetch() trabaja con peticiones HTTP, por lo tanto, si se realiza una aplicación web con esta función, se la debe probar a través de un servidor web. Esta prueba nunca se debe hacer al abrir de forma local el documento HTML en una pestaña del navegador web, porque fetch() no funcionará.

Figura 63: Resultado de la petición `fetch` al servicio de JSONPlaceholder



```

> fetch('https://jsonplaceholder.typicode.com/posts')
  .then((response) => response.json())
  .then((data) => console.table(data))
  .catch((error) => console.error(error));
<  Promise {<pending>}

```

(índice)	userId	id	title	body
0	1	1	'sunt aut facere ...'	'quia et suscipit\nsuscipit recusandae...
1	1	2	'qui est esse'	'est rerum tempore vitae\nsequi sint n...
2	1	3	'ea molestias qua...	'et iusto sed quo iure\\nvoluptatem occ...
3	1	4	'eum et est occae...	'ullam et saepe reiciendis voluptatem ...
4	1	5	'nesciunt quas od...	'repudiandae veniam quaerat sunt sed\\n...
5	1	6	'dolorem eum magn...	'ut aspernatur corporis harum nihil qu...
6	1	7	'magnam facilis a...	'dolore placeat quibusdam ea quo vitae...
7	1	8	'dolorem dolore e...	'dignissimos aperiam dolorem qui eum\\n...
8	1	9	'nesciunt iure om...	'consectetur animi nesciunt iure dolor...
9	1	10	'optio molestias ...'	'quo et expedita modi cum officia vel ...'
10	2	11	'et ea vero quia ...'	'delectus reiciendis molestiae occaecata...

Fuente: elaboración propia.

## Peticiones

La petición anteriormente realizada corresponde a una petición predeterminada, conocida como `get`. Fetch también soporta otro tipo de peticiones, como ser `post`, `put`, `delete` o `patch`. A continuación, se muestran algunos ejemplos de estas últimas. En estos casos, fetch requerirá un segundo parámetro.

## **Post**

**Figura 64: Post**



```
fetch('https://jsonplaceholder.typicode.com/posts', {
  method: 'POST',
  body: JSON.stringify({
    title: 'nuevo Post',
    body: 'Esta es una nueva noticia para mi blog.',
    userId: 1,
  }),
  headers: { 'Content-type': 'application/json; charset=UTF-8' },
})
.then((response) => response.json())
.then((data) => console.log(data));
```

Fuente: elaboración propia.

Para crear un nuevo recurso en el servidor, *fetch* espera un segundo parámetro. Este segundo parámetro representará un objeto literal, que debe disponer de: la información a crear en el servidor, el tipo de datos que le enviamos al servidor, y qué método utilizaremos para crear dicho recurso.

La propiedad **method** nos permite definir que realizaremos un método del tipo **post**.

La propiedad **body** nos permite definir, en este caso, la estructura del recurso que crearemos en el servidor (para nosotros, un objeto literal).

La propiedad **headers** espera, al menos, la información correspondiente al tipo de contenido que enviamos al servidor. En nuestro caso, contenido en formato JSON con el juego de caracteres UTF-8.

El método **.then()** nos permite controlar la posible respuesta del servidor. Generalmente, los servidores retornan un código de estado, en nuestro caso debería ser **201**. Algunas veces retornan también la misma estructura del recurso, para que validemos que lo que recibió el servidor es lo correcto.

## **Put**

**Figura 65: Put**



```
fetch('https://jsonplaceholder.typicode.com/posts/1', {
  method: 'PUT',
  body: JSON.stringify({
    id: 1,
    title: 'Cambio titulo del post',
    body: 'Aquí defino el texto o cuerpo de la noticia a modificar',
    userId: 1,
  }),
  headers: { 'Content-type': 'application/json; charset=UTF-8' },
})
.then((response) => response.json())
.then((data) => console.log(data));
```

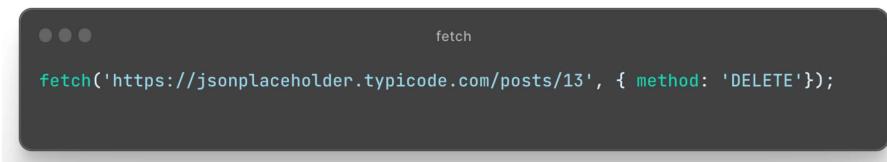
Fuente: elaboración propia.

La estructura de una petición para modificar un recurso de servidor es casi igual a la petición post, con la diferencia de que el método a utilizar es **put**. Otra pequeña diferencia que encontramos en este código es que la URL a la cual peticionamos recibe como parámetro el **ID**, o identificador, del recurso que deseamos modificar en el servidor.

El manejo de las respuestas de servidor se realiza, como siempre, con los métodos **.then()** y **.catch()**, en el caso de querer manejar algún posible error.

### Delete

Figura 66: Delete



```
fetch('https://jsonplaceholder.typicode.com/posts/13', { method: 'DELETE'});
```

Fuente: elaboración propia.

Usualmente, es el método más simple, dado que elimina un recurso del servidor. En este caso, **fetch()** solo espera como parámetro adicional el método en cuestión, y no requiere que le indiquemos un *body*.

En algunos casos, el método *delete* puede retornarnos la estructura completa del recurso eliminado. En otros casos, puede retornar un *array* vacío [] o una estructura de objeto literal vacía {}. Siempre se debe buscar este tipo de respuestas en la documentación oficial del servicio API REST que se utilice.

### Manejo de códigos de estado con **fetch**

Si queremos controlar los códigos de estado de servidor desde nuestra lógica de JS, podemos recurrir a utilizar la respuesta del servidor, **response**, que pasamos como parámetro al primer método **.then()** que controla nuestro código.

Dentro de **response** encontraremos muchas propiedades. Entre ellas, **status**, que contiene la respuesta obtenida por parte del servidor. Si queremos manejar en profundidad los códigos de estado con **fetch**, podemos preguntar por el número de error exitoso que esperamos, o definir un error personalizado, si no logramos capturar el recurso de servidor de manera satisfactoria.

Figura 67: Respuesta del servidor



```
fetch('https://jsonplaceholder.typicode.com/posts')
  .then(response => {
    return response.status === 200
      ? response.json()           //solicitud exitosa
      : throw new Error('Error al obtener los datos'); //arroja un error personalizado
  })
  .then(data => console.table(data))
  .catch(error => console.error(error));
```

Fuente: elaboración propia.

### Actividad 8

Acceder a la guía de JSONPlaceHolder: <https://jsonplaceholder.typicode.com/guide/>.

Ubicar el código de ejemplo para generar un post nuevo (método post).

Copiar el código de ejemplo y pegarlo en la consola JS de DevTools. Se lo debe modificar antes de ejecutarlo.

Definir el siguiente título y cuerpo de un nuevo post.

Title: "Post generado por mí".

Body: "Este es un post generado a partir de la microactividad 8".

Modificar el último método de control .then(): pasar datos como parámetro y ejecutar console.table() para visualizar la respuesta del servidor.

Para conocer la resolución de todas las actividades, descargá el siguiente PDF:



Fuente: elaboración propia.

## **Video de habilidades**

## **Glosario**

## **Referencias**

**Microsoft** (2023). *VS Code [software]*. Redmond, Washington.