

JS



JAVASCRIPT – parte 2

Índice

- **Introducción a Eventos**
- **Trabajando con Eventos**
- **BOM y Objetos nativos**
- **Formularios y expresiones regulares**
- **Callbacks y Promesas**
- **AJAX**
- **Axios**
- **Web Storage**
- **API's – Drag & Drop**



desmotivaciones.es

Estudiar debes

si buenas notas sacar quieres.

EVENTOS

Un evento es cualquier cosa que sucede en nuestro documento.

- El contenido se ha leído
- El contenido se ha cargado
- El usuario mueve el ratón
- El usuario pulsa una tecla
- La ventana se ha cerrado
- Y un largo etc.

~~`<p onclick="saludo()">Soy un párrafo</p>`~~

`<p (click)="saludo()">Soy un párrafo</p>`

No hacerlo por varios motivos:

- Para no mezclar HTML y Javascript.
- Difícil ejecutar más de un evento de esta forma.



ANGULAR

`Element.addEventListener('event', callback)`



CORRECTO

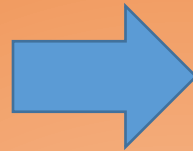
LISTADO DE EVENTOS POSIBLES

<https://developer.mozilla.org/es/docs/Web/Events>

- Los eventos que están en color azul son aquellos que funcionan y no tienen ningún error, por ejemplo “click”.
- Los de color rojo son aquellos que presentan algún error que no se ha corregido
- Los del pulgar hacia abajo están obsoletos y no se utilizan.

index.html

```
<button id="button">Click Me!</button>  
  
<script src="scripts.js"></script>
```



scripts.js

```
const button = document.getElementById('button')  
  
button.addEventListener('click', () => {  
  console.log('CLICK');  
})
```

Función de flecha para indicar lo que queremos que ocurra cuando suceda el evento

EVENTOS QUE VAMOS A TRABAJAR

Eventos de ratón:

click - cuando pulsamos el botón izquierdo del ratón

dblclick - cuando pulsamos dos veces seguidas el botón izquierdo del ratón

mouseenter - cuando entramos en la zona que tiene el evento

mouseleave - cuando salimos de la zona que tiene el evento

mousedown - cuando pulsamos el botón izquierdo del ratón

mouseup - cuando soltamos el botón izquierdo del ratón

mousemove - cuando movemos el ratón

Eventos de teclado:

keydown - cuando pulsamos una tecla

keyup - cuando soltamos una tecla

keypress - cuando pulsamos una tecla y no la soltamos

EJERCICIO BÁSICO 1

- Crea una página **index.html** que llame a un archivo javascript cuyo nombre sea **scripts.js** y una hoja de estilo cuyo nombre sea **styles.css**
- En el archivo index.html crea un div que tenga una clase que se llame box y cuyo id se llame box. En la hoja de estilo defina esta clase con los siguientes valores:
 - width: 100px
 - height: 100px
 - background: red
- Cuando el ratón entre dentro de la caja, se cambiará el color a verde.
- Cuando el ratón salga de la caja el color deberá cambiar a rojo.
- Cuando pulsemos el botón izquierdo del ratón estando situados sobre la caja, aparecerá por consola el mensaje “Has pulsado la caja.
- Al soltar el botón izquierdo del ratón en la caja, aparecerá por consola el mensaje “Has soltado el botón izquierdo dentro de la caja”.

EJERCICIO BÁSICO 2

- Sobre el ejercicio anterior añade un input de tipo texto.
 - Al pulsar una tecla deberá aparecer el mensaje por consola “Has pulsado una tecla”
 - Al soltar la tecla deberá aparecer el mensaje por consola “Has soltado una tecla”

¿Cómo podemos saber qué tecla concreta se ha pulsado?, investiga que tendrías que utilizar para detectar la tecla concreta pulsada y muéstrala en el mensaje por consola.

EJERCICIO BÁSICO 3

- Crea un formulario con un input de tipo texto y un botón “Enviar”. Al pulsar el botón. Crea un evento para que al soltar una tecla se lance una función que vaya mostrando por consola todo lo que se escribe en el input.

¿cómo podemos sacar información del evento? Averigua como podemos obtener la tecla concreta que se ha pulsado cada vez.

¿Cómo accedemos a la información del evento?

Ejemplo de acceso al target del evento:

```
button.addEventListener('click', (e) => {  
  console.log(e.target);  
})
```

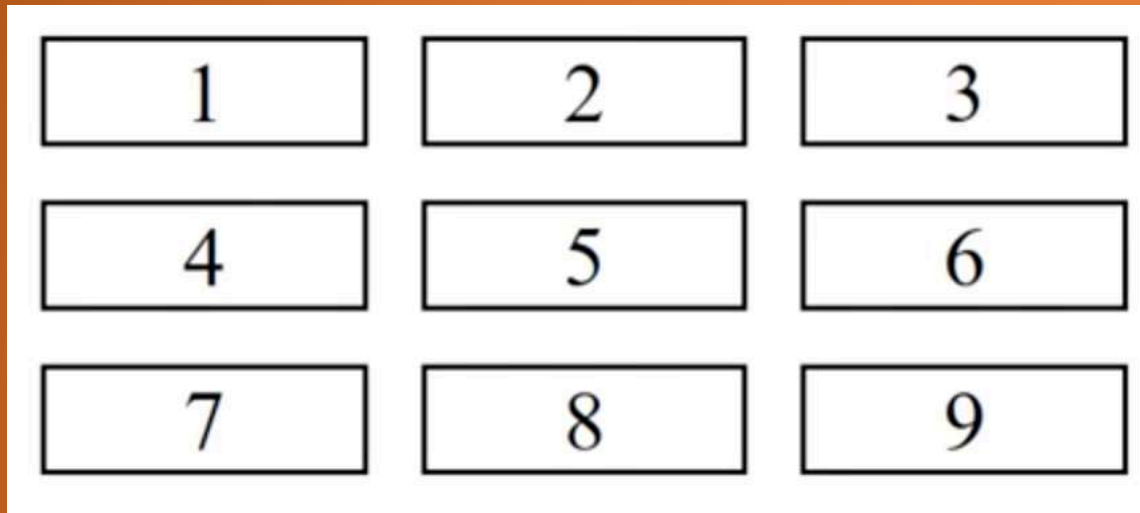


Se puede encontrar mucha información del evento dentro del target
Por ejemplo:

- Clases que tiene el botón
- Posición del botón
- Si está activado o no
- Si es arrastrable o no
- Tipo de nodo
- Quién es su padre e información del su nodo padre
- Propiedades CSS que tiene el botón
- Tipo de botón que es
- Etc

Delegación de eventos. Optimización de recursos

¿Dada la siguiente galería, como harías para añadir un evento a cada uno de los div y saber en cual se ha hecho click?



```
<div id="gallery" class="gallery">
  <div class="gallery__item">1</div>
  <div class="gallery__item">2</div>
  <div class="gallery__item">3</div>
  <div class="gallery__item">4</div>
  <div class="gallery__item">5</div>
  <div class="gallery__item">6</div>
  <div class="gallery__item">7</div>
  <div class="gallery__item">8</div>
  <div class="gallery__item">9</div>
</div>
```

Delegación de eventos. Optimización de recursos

Una opción podría ser localizar la galería con `document.getElementById("gallery")` y hacer un bucle `for` y a cada uno de los botones le añadimos un evento.

Esto consume muchos recursos del navegador.

Para solucionarlo ponemos una escucha al gallery container y localizar en cual de los hijos se ha hecho el click. A continuación podemos ver como hacerlo.


```
const gallery = document.getElementById('gallery')

gallery.addEventListener('click', (e) => {
  console.log(e.target.textContent);
})
```

Esto además de utilizarse en galerías, se puede utilizar también en formularios, cuando hay muchos campos es mucho más óptimo hacerlo de esta forma

Eventos y Formularios

¿Cómo puedo hacer para que al pulsar un botón de un formulario de tipo **submit** no se recargue la página?



```
form.addEventListener('submit', (e) => {  
  e.preventDefault()  
  console.log('El formulario se ha enviado');  
})
```

Con **preventDefault** conseguimos que no se ejecute el comportamiento por defecto. No sirve solo para formularios sino para cualquier elemento de HTML que tenga un comportamiento determinado.

```
const link = document.getElementById('link')  
  
link.addEventListener('click', (e) => {  
  e.preventDefault()  
})
```


Eventos y Formularios. Disparar eventos

Podemos disparar eventos sin necesidad de interacción con el usuario.

Para hacerlo pongo el `elemento.eventoadisparar()`. Por ejemplo si tengo una constante `button`, para lanzar el evento `click` sin que el usuario tenga que hacer click en ningún sitio, sería suficiente escribiendo en mi código `button.click()`



BOM o Browser Object Model

Browser Object Model o BOM, que permite acceder y modificar las propiedades de las ventanas del propio navegador.

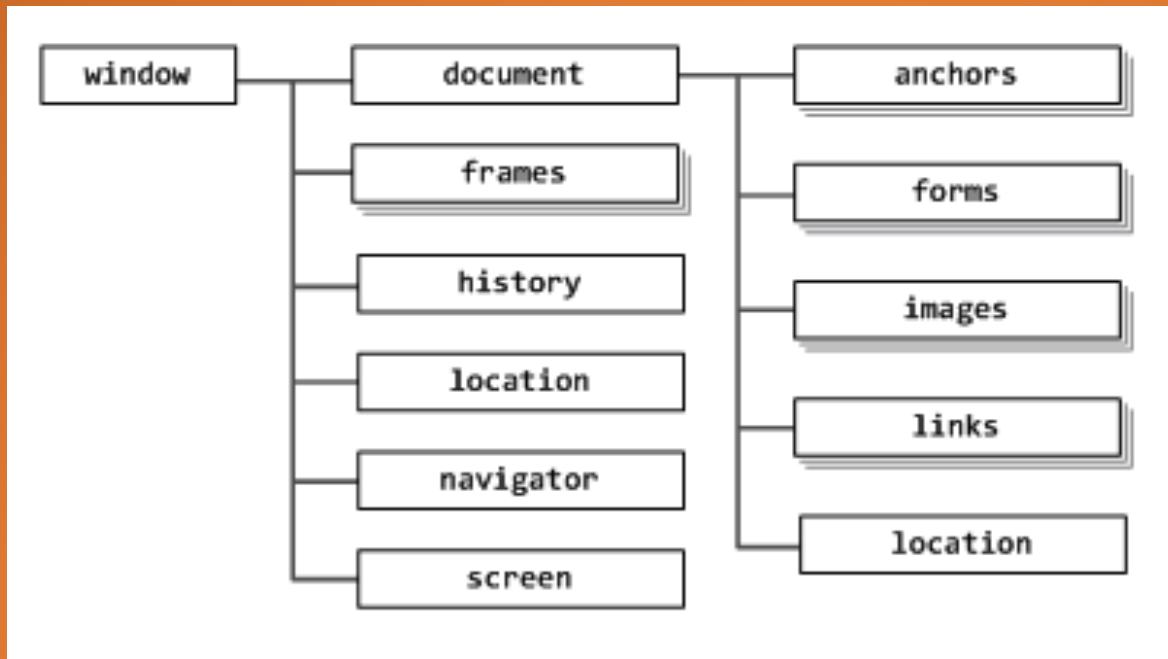
Mediante BOM, es posible redimensionar y mover la ventana del navegador, modificar el texto que se muestra en la barra de estado y realizar muchas otras manipulaciones no relacionadas con el contenido de la página HTML.

Algunos de los elementos que forman el BOM son los siguientes:

- Crear, mover, redimensionar y cerrar ventanas de navegador.
- Obtener información sobre el propio navegador.
- Propiedades de la página actual y de la pantalla del usuario.
- Gestión de cookies.
- Objetos ActiveX en Internet Explorer.

BOM o Browser Object Model

El BOM está compuesto por varios objetos relacionados entre sí. El siguiente esquema muestra los objetos de BOM y su relación:



Los objetos mostrados con varios recuadros superpuestos son arrays. El resto de objetos, representados por un rectángulo individual, son objetos simples. En cualquier caso, todos los objetos derivan del objeto **window**.

Objetos nativos

Son objetos que ya están predefinidos y que por tanto podemos utilizarlos. Son:

- **Objeto window:** representa la ventana completa del navegador. Mediante este objeto, es posible mover, redimensionar y manipular la ventana actual del navegador. Incluso es posible abrir y cerrar nuevas ventanas de navegador.
- **Objeto document:** es el único que pertenece tanto al DOM (como se vio en el capítulo anterior) como al BOM. Desde el punto de vista del BOM, el objeto document proporciona información sobre la propia página HTML.
- **Objeto location:** es una propiedad tanto del objeto window como del objeto document. Representa la URL de la página HTML que se muestra en la ventana del navegador y proporciona varias propiedades útiles para el manejo de la URL.
- **Objeto navigator:** permite obtener información sobre el propio navegador.
- **Objeto screen:** se utiliza para obtener información sobre la pantalla del usuario.
- **Objeto history:** trabajar con el historial de la ventana que estamos trabajando.

A continuación vamos a ver en detalle cada uno de estos objetos

Objetos nativos. Objeto window

- Es el objeto global del que descienden todos los objetos.
- Representa la ventana completa del navegador. Mediante este objeto, es posible mover, redimensionar y manipular la ventana actual del navegador. Incluso es posible abrir y cerrar nuevas ventanas de navegador.

BOM define cuatro métodos para manipular el tamaño y la posición de la ventana:

- `moveBy(x, y)` desplaza la posición de la ventana x píxel hacia la derecha y y píxel hacia abajo. Se permiten desplazamientos negativos para mover la ventana hacia la izquierda o hacia arriba.
- `moveTo(x, y)` desplaza la ventana del navegador hasta que la esquina superior izquierda se encuentre en la posición (x, y) de la pantalla del usuario. Se permiten desplazamientos negativos, aunque ello suponga que parte de la ventana no se visualiza en la pantalla.
- `resizeBy(x, y)` redimensiona la ventana del navegador de forma que su nueva anchura sea igual a $(\text{anchura_anterior} + x)$ y su nueva altura sea igual a $(\text{altura_anterior} + y)$. Se pueden emplear valores negativos para reducir la anchura y/o altura de la ventana.
- `resizeTo(x, y)` redimensiona la ventana del navegador hasta que su anchura sea igual a x y su altura sea igual a y . No se permiten valores negativos.

Objetos nativos. Objeto window

Los navegadores son cada vez menos permisivos con la modificación mediante JavaScript de las propiedades de sus ventanas. De hecho, la mayoría de navegadores permite a los usuarios bloquear el uso de JavaScript para realizar cambios de este tipo. De esta forma, una aplicación nunca debe suponer que este tipo de funciones están disponibles y funcionan de forma correcta.

A continuación se muestran algunos ejemplos de uso de estas funciones:

```
// Mover la ventana 20 píxel hacia la derecha y 30 píxel hacia abajo  
window.moveBy(20, 30);  
  
// Redimensionar la ventana hasta un tamaño de 250 x 250  
window.resizeTo(250, 250);  
  
// Agrandar la altura de la ventana en 50 píxel  
window.resizeBy(0, 50);  
  
// Colocar la ventana en la esquina izquierda superior de la ventana  
window.moveTo(0, 0);
```


Objetos nativos. Objeto document

El objeto `document` es el único que pertenece tanto al DOM (como se vio en el tema anterior) como al BOM. Desde el punto de vista del BOM, el objeto `document` proporciona información sobre la propia página HTML.

Algunas de las propiedades más importantes definidas por el objeto `document` son:

Propiedad	Descripción
<code>lastModified</code>	La fecha de la última modificación de la página
<code>referrer</code>	La URL desde la que se accedió a la página (es decir, la página anterior en el array <code>history</code>)
<code>title</code>	El texto de la etiqueta <code><title></code>
<code>URL</code>	La URL de la página actual del navegador

Las propiedades `title` y `URL` son de lectura y escritura, por lo que además de obtener su valor, se puede establecer de forma directa

```
// modificar el título de la página
document.title = "Nuevo titulo";

// llevar al usuario a otra página diferente
document.URL = "http://nueva_pagina";
```

Objetos nativos. Objeto document

Además de propiedades, el objeto `document` contiene varios `arrays` con información sobre algunos elementos de la página:

Array	Descripción
<code>anchors</code>	Contiene todas las "anclas" de la página (los enlaces de tipo <code></code>)
<code>applets</code>	Contiene todos los applets de la página
<code>embeds</code>	Contiene todos los objetos embebidos en la página mediante la etiqueta <code><embed></code>
<code>forms</code>	Contiene todos los formularios de la página
<code>images</code>	Contiene todas las imágenes de la página
<code>links</code>	Contiene todos los enlaces de la página (los elementos de tipo <code></code>)

Objetos nativos. Objeto location

El objeto `location` es uno de los objetos más útiles del BOM. Debido a la falta de estandarización, `location` es una propiedad tanto del objeto `window` como del objeto `document`.

El objeto `location` representa la URL de la página HTML que se muestra en la ventana del navegador y proporciona varias propiedades útiles para el manejo de la URL:

Propiedad	Descripción
hash	El contenido de la URL que se encuentra después del signo # (para los enlaces de las anclas) <code>http://www.ejemplo.com/ruta1/ruta2/pagina.html#seccion</code> hash = #seccion
host	El nombre del servidor <code>http://www.ejemplo.com/ruta1/ruta2/pagina.html#seccion</code> host = www.ejemplo.com
hostname	La mayoría de las veces coincide con host, aunque en ocasiones, se eliminan las www del principio <code>http://www.ejemplo.com/ruta1/ruta2/pagina.html#seccion</code> hostname = www.ejemplo.com
href	La URL completa de la página actual <code>http://www.ejemplo.com/ruta1/ruta2/pagina.html#seccion</code> URL = <code>http://www.ejemplo.com/ruta1/ruta2/pagina.html#seccion</code>
pathname	Todo el contenido que se encuentra después del host <code>http://www.ejemplo.com/ruta1/ruta2/pagina.html#seccion</code> pathname = /ruta1/ruta2/pagina.html
port	Si se especifica en la URL, el puerto accedido <code>http://www.ejemplo.com:8080/ruta1/ruta2/pagina.html#seccion</code> port = 8080 La mayoría de URL no proporcionan un puerto, por lo que su contenido es vacío <code>http://www.ejemplo.com/ruta1/ruta2/pagina.html#seccion</code> port = (vacío)
protocol	El protocolo empleado por la URL, es decir, todo lo que se encuentra antes de las dos barras inclinadas // <code>http://www.ejemplo.com/ruta1/ruta2/pagina.html#seccion</code> protocol = http:
search	Todo el contenido que se encuentra tras el símbolo ?, es decir, la consulta o "query string" <code>http://www.ejemplo.com/pagina.php?variable1=valor1&variable2=valor2</code> search = ?variable1=valor1&variable2=valor2

Objetos nativos. Objeto location

De todas las propiedades, la más utilizada es `location.href`, que permite obtener o establecer la dirección de la página que se muestra en la ventana del navegador.

Además de las propiedades de la tabla anterior, el objeto `location` contiene numerosos métodos. Algunos de los métodos más útiles son los siguientes:

```
// Método assign()
location.assign("http://www.ejemplo.com"); // Equivalente a location.href = "http://ww
w.ejemplo.com"

// Método replace()
location.replace("http://www.ejemplo.com");
// Similar a assign(), salvo que se borra la página actual del array history del navega
dor

// Método reload()
location.reload(true);
/* Recarga la página. Si el argumento es true, se carga la página desde el servidor.
   Si es false, se carga desde la cache del navegador */
```

Más información sobre `location`: <https://developer.mozilla.org/es/docs/Web/API/Location>

Objetos nativos. Objeto navigator

El objeto `navigator` es uno de los primeros objetos que incluyó el BOM y permite obtener información sobre el propio navegador. En Internet Explorer, el objeto `navigator` también se puede acceder a través del objeto `clientInformation`.

Aunque es uno de los objetos menos estandarizados.

El objeto `navigator` se emplea habitualmente para detectar el tipo y/o versión del navegador en las aplicaciones cuyo código difiere para cada navegador. Además, se emplea para detectar si el navegador tiene habilitadas las cookies y Java y también para comprobar los plugins disponibles en el navegador.

Mas información sobre navigator: <https://developer.mozilla.org/es/docs/Web/API/Navigator>

Objetos nativos. Objeto screen

El objeto `screen` se utiliza para obtener información sobre la pantalla del usuario. Uno de los datos más importantes que proporciona el objeto `screen` es la resolución del monitor en el que se están visualizando las páginas. Los diseñadores de páginas web necesitan conocer las resoluciones más utilizadas por los usuarios para adaptar sus diseños a esas resoluciones. Las siguientes propiedades están disponibles en el objeto `screen`:

Propiedad	Descripción
<code>availHeight</code>	Altura de pantalla disponible para las ventanas
<code>availWidth</code>	Anchura de pantalla disponible para las ventanas
<code>colorDepth</code>	Profundidad de color de la pantalla (32 bits normalmente)
<code>height</code>	Altura total de la pantalla en píxel
<code>width</code>	Anchura total de la pantalla en píxel

Mas información sobre `screen`: <https://developer.mozilla.org/es/docs/Web/API/Screen>

Objetos nativos. Objeto screen

La altura/anchura de pantalla disponible para las ventanas es menor que la altura/anchura total de la pantalla, ya que se tiene en cuenta el tamaño de los elementos del sistema operativo como por ejemplo la barra de tareas y los bordes de las ventanas del navegador. Además de la elaboración de estadísticas de los equipos de los usuarios, las propiedades del objeto `screen` se utilizan por ejemplo para determinar cómo y cuanto se puede redimensionar una ventana y para colocar una ventana centrada en la pantalla del usuario. El siguiente ejemplo redimensiona una nueva ventana al tamaño máximo posible según la pantalla del usuario:

```
window.moveTo(0, 0);  
window.resizeTo(screen.availWidth, screen.availHeight);
```

Mas información sobre screen: <https://developer.mozilla.org/es/docs/Web/API/Screen>

Objetos nativos. Objeto history

El objeto `history` nos permite trabajar con el historial de la ventana que estamos trabajando.

Tiene 3 métodos importantes:

- `back()`: navegar hacia atrás
- `forward()`: navegar hacia adelante
- `go (n| -n)`: recibe un número positivo o un número negativo como parámetro. Navega a través del historial X páginas en función del valor que indiquemos por parámetro.

Tiene la propiedad `length`

```
>> history
<  History
    length: 5
    scrollRestoration: "auto"
    state: null
```

En este ejemplo al escribir por consola el valor de `history` vemos que su propiedad `length = 5`. Indica que tiene 5 páginas en el historial

Más información sobre `history`: https://developer.mozilla.org/es/docs/DOM/Manipulando_el_historial_del_navegador

Objetos nativos. Objeto date

El objeto date tiene una serie de métodos que se pueden consultar aquí:

- https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Date
- https://www.w3schools.com/jsref/jsref_obj_date.asp

El objeto date tiene la peculiaridad de que necesitamos instanciarlo para usarlo. Por ejemplo:

```
const date = new Date()  
  
console.log(date.getDay())
```

Cuidado que la fecha empieza por 0, por tanto el 2 según el calendario anglosajón sería Martes. Igual pasa con los meses que empieza a contar por 0. Por tanto Enero sería 0.

TIMERS

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

- **setTimeout(()=>{code}, delay-in-miliseconds):**

Hace que se ejecute la función despues de delay. Si lo referenciamos mediante una variable/constante podemos pararlo con clearTimeout(referencia)

Si queremos hacerlo como función interna.

```
button.addEventListener('click', () => {  
  //setTimeout(saludar,3000)  
  const timeout = setTimeout(() => {  
    console.log('ADIOS');  
  }, 3000)  
  
  clearTimeout(timeout)  
})  
  
const saludar = () => {  
  console.log('Hola')  
}
```

Si queremos hacerlo como función externa.

TIMERS

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

setInterval(()=>{code}, delay-in-miliseconds) :

Hace que se ejecute la función cada delay milisegundos. Si lo referenciamos mediante una variable/constante podemos pararlo con **clearInterval(referencia)**

Ejemplo de contador de segundos:

```
setInterval(() => {  
  console.log(cont)  
  cont++  
}, 1000)
```

Si queremos pararlo funcionaría igual que el Timeout

EJERCICIOS – Date y Timers

Ejercicio 1: Crea una página que muestre por pantalla un cronómetro y la fecha actual. El formato que deberá aparecer es lo siguiente:

Hoy es 30 - 9 - 2019 y son las 21:4:23 horas

Ejercicio 2: Desarrolla una aplicación web que funcione de alarma. Mostrar hora actual y preguntar al usuario a qué hora sonará. Una vez que llegue a la hora introducida por el usuario, debe preguntarle si quiere descansar un poco más y vuelva a mostrar el mensaje a los dos minutos.

FORMULARIOS. Validación de formularios

Es muy importante validar los datos tanto en el FRONTEND como en el BACKEND

Distintas formas de validar formularios:

- Crear un objeto con los datos del formulario (NO ES LA IDEAL) aunque según la complejidad del formulario puede servir.
- Utilizando expresiones regulares

FORMULARIOS. Validación de formularios

Ejemplo de cómo sería la validación con la creación de un objeto.

Forms I

Name:

E-mail:

Male

☐

Female

☐

I accept the terms and conditions

☐

Send

```
const form = document.getElementById('form')
const button = document.getElementById('submitButton')

const name = document.getElementById('name')
const email = document.getElementById('email')
const gender = document.getElementById('gender')
const terms = document.getElementById('terms')

const formIsValid = {
  name: false,
  email: false,
  gender: false,
  terms: false
}

form.addEventListener('submit', (e) => {
  e.preventDefault()
  validateForm()
})

name.addEventListener('change', (e) => {
  if(e.target.value.trim().length > 0) formIsValid.name = true
})

email.addEventListener('change', (e) => {
  if(e.target.value.trim().length > 0) formIsValid.email = true
})

gender.addEventListener('change', (e) => {
  console.log(e.target.checked)
  if(e.target.checked === true) formIsValid.gender = true
})

terms.addEventListener('change', (e) => {
  formIsValid.terms = e.target.checked
  e.target.checked ? button.removeAttribute('disabled') : button.setAttribute('disabled', true)
})

const validateForm = () => {
  const formValues = Object.values(formIsValid)
  const valid = formValues.findIndex(value => value === false)
  if(valid === -1) form.submit()
  else alert('Form invalid')
}
```


FORMULARIOS y Expresiones Regulares

Expresiones regulares: Son una secuencia de caracteres que forma un patrón de búsqueda, principalmente utilizada para la búsqueda de patrones de cadenas de caracteres u operaciones de sustituciones.

¿Cómo puedo probar una expresión regular?

<https://regex101.com/>

Información sobre expresiones regulares

https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Regular_Expressions

¿Cuál es la sintaxis?

/ patrón/banderas

¿Cuáles son las banderas?

i: ignore case. No diferencia entre mayúsculas y minúsculas

g: global. Busca de forma global, es decir, no se para después de la primera coincidencia

FORMULARIOS y Expresiones Regulares

Ejemplo muy básico de utilización de expresiones regulares

```
const text = document.getElementById('text').textContent
const regex = /lorem/gi

console.log(regex.test(text))
```

Otra forma posible de crear expresiones regulares

```
const text = document.getElementById('text').textContent
const regex = /lorem/gi
const regex2 = new RegExp('lorem', 'gi')
const regex3 = new RegExp('/lorem/', 'gi')

console.log(regex2.test(text))
```

FORMULARIOS y Expresiones Regulares

COMODINES

- **Sustitución:** Define un comodín dentro del patrón. El símbolo es el "."
- **Listado de caracteres válidos:** Entre corchetes se pone una lista de los caracteres válidos. Por ejemplo `[aeiou]` Con esto cogeríamos todas las vocales
- **Rangos:** Entre corchetes si ponemos un guion entre dos caracteres establecemos un rango. `[a-z]` Todas las letras minúsculas. Tabla ASCII <https://ascii.cl/es/>
- **Mezcla entre rangos y listas:** Podemos unir los dos anteriores en una sola expresión. Por ejemplo `[0-5ou]` Serían números del 0 al 5, la letra "o" y la letra "u"
- **Cadenas completas:** Para establecer una cadena completa debe ir entre paréntesis, si queremos más palabras irán separadas por un pipe. Por ejemplo: `(lorem|amet)` es válida la palabra "lorem" y la palabra "amet".

FORMULARIOS y Expresiones Regulares

DELIMITADORES

- **^** Antes de este símbolo no puede haber nada
- **\$** Después de este símbolo no puede haber nada. Por ejemplo **^hola\$**

CANTIDAD:

- **llaves:** lo que está antes tiene que aparecer la cantidad exacta de veces. Hay tres combinaciones posibles.
 - {n}** Se tiene que repetir n veces
 - {n,m}** Se tiene que repetir entre n y m veces, ambas incluidas.
 - {n,}** Se tiene que repetir como mínimo n veces y sin máximo.Por ejemplo **^[a-zA-Z]{1,3}@{1}\$**
- **asterisco:** Lo que está antes del asterisco puede estar, puede no estar y se puede repetir. **.*@.*\..***
- **interrogación:** Lo que está antes de la interrogación puede no estar, pero si está solo puede aparecer una vez. Por ejemplo **^[ae]?\$**
- **operador +:** lo que está antes del + tiene que estar una vez como mínimo. Por ejemplo: **A-[0-9]+**

FORMULARIOS y Expresiones Regulares

CARACTERES ESPECIALES

- **\s**: Coincide con un carácter de espacio, entre ellos incluidos espacio, tab, salto de página, salto de línea y retorno de carro. *Ejemplo de 2 palabras con un espacio en medio* **$^[a-zA-Z]+\s[a-zA-Z]+$$**
- **\S**: Coincide con todo menos caracteres de espacio. *Ejemplo: palabra de 5 letras con ningún espacio* **$^\S{5}$$**
- **\d**: Coincide con un carácter de número. Equivalente a [0-9]. *Ejemplo de 5 dígitos* **$^\d{5}$$**
- **\D**: Coincide con cualquier carácter no numérico. Equivalente a [^0-9]. *Por ejemplo* **$^\D{5}$$**
- **\w**: Coincide con cualquier carácter alfanumérico, incluyendo el guión bajo. Equivalente a [A-Za-z0-9_]. *Por ejemplo* **$^\w+@$$**
- **\W**: Coincide con todo menos caracteres de palabra. *Por ejemplo* **$^\W+$$**

EJERCICIOS EXPRESIONES REGULARES

Crea un formulario en el que es necesario que los siguientes aspectos sean validados con expresiones regulares:

- DNI
- Email
- Cuenta bancaria. De la cuenta bancaria validaremos el IBAN
- Complejidad de la password
- Nombre de usuario
- URL
- IP
- Tarjeta de crédito.

Para la validación se proporciona un archivo **validate.js** que contiene funciones de validación con expresiones regulares que deberas utilizar en tu formulario.

CALLBACK

- Es una función que se ejecuta dentro de otra función.
- Los callback no son asíncronos es decir se ejecutan por orden.

```
const getUser = (id, cb) => {  
  const user = {  
    name: 'Dorian',  
    id: id  
  }  
  
  if (id == 2) cb('User not exist')  
  else cb(null, user)  
}  
  
getUser(1, (err, user) => {  
  if (err) return console.log(err)  
  console.log(`User name is ${user.name}`);  
})
```

En este ejemplo **cb** es un callback que se pasa como parámetro de la función `getUser`

EJERCICIO Callback

- Crea un array de objetos usuario. Cada objeto usuario tiene su id y su nombre. El array contendrá al menos 3 usuarios. Con esto vamos a simular que tenemos una base de datos de usuarios.
- Crea una función que solicite el usuario y recibirá por parámetro el id y una función de callback. Localizará al usuario sin necesidad de hacer ningún bucle y pintará por pantalla el nombre del usuario. Si el usuario no existe se ejecutará el callback indicando que “Not exist a user with id XXX”. Si existe se ejecutará el callback sin error y devolvemos el usuario.

Utiliza las funciones de flechas.

PROMESAS

- Es un objeto que representa un valor que puede que este disponible ahora, o en el futuro o quizás nunca. Como no sabemos cuando estará disponible ese valor, hay que posponerlo en el tiempo.

Por ejemplo una petición HTTP que nos devuelva un JSON con datos y no sabemos cuanto tiempo va a durar. La promesa realizará la operación asíncrona, el código seguirá ejecutándose pero en la promesa hemos dejado el código que queremos que se ejecute cuando la promesa termine.

- Este objeto tiene dentro 2 callbacks internos.
- Para crear una promesa: **const promise = new Promise ((resolve, reject)=>{})** los parámetros resolve y reject son callbacks obligatorios ya vienen preprogramados. El callback resolve se ejecuta cuando todo va bien y reject cuando no va bien.

Mas info: https://www.youtube.com/watch?v=FK5-YW_-Gc0

PROMESAS

- Es un objeto que representa un valor que puede estar disponible ahora, o en el futuro o quizás nunca. que dentro tiene 2 callbacks internos.
- Para crear una promesa: **const promise = new Promise ((resolve, reject)=>{})** los parámetros resolve y reject son callbacks obligatorios ya vienen preprogramados. El callback resolve se ejecuta cuando todo va bien y reject cuando no va bien.

```
const getUser = (id, cb) => {  
  const user = {  
    name: 'Dorian',  
    id: id  
  }  
  
  if (id == 2) cb('User not exist')  
  else cb(null, user)  
}  
  
getUser(1, (err, user) => {  
  if (err) return console.log(err)  
  console.log(`User name is ${user.name}`);  
})
```



```
const getUser = (id) => {  
  const user = users.find(user => user.id == id)  
  const promise = new Promise((resolve, reject) => {  
    if (!user) reject(`Not exist a user with id ${id}`)  
    else resolve(user)  
  })  
  
  return promise  
}
```

El ejemplo visto anteriormente sería así utilizando **PROMESAS**

PROMESAS. EJEMPLO

Supongamos que vamos a comprar comida a un restaurante de comida rápida, cuando terminamos de pagar por nuestra comida nos dan un **ticket con un número**, cuando llamen a ese número podemos entonces ir a buscar nuestra comida.

Ese ticket que nos dieron es nuestra promesa, ese ticket nos indica que eventualmente vamos a tener nuestra comida, pero que todavía no la tenemos. Cuando llaman a ese número para que vayamos a buscar la comida entonces quiere decir que la promesa se completó. Pero resulta que una promesa se puede completar correctamente o puede ocurrir un error, ¿Qué error puede ocurrir en nuestro caso? Por ejemplo puede pasar que el restaurante no tenga más comida, entonces cuando nos llamen con nuestro número pueden pasar dos cosas:

1. Nuestro pedido se resuelve y obtenemos la comida.
2. Nuestro pedido es rechazado y obtenemos una razón del por qué.

Pongamos esto en código:

```
const ticket = getFood();  
  
ticket  
  .then(food => eatFood(food))  
  .catch(error => getRefund(error));
```

PROMESAS. EJEMPLO

Cuando tratamos de obtener la comida (`getFood`) obtuvimos una promesa (`ticket`), si esta se resuelve correctamente entonces recibimos nuestra comida (`food`) y nos la comemos (`eatFood`). Si nuestro pedido es rechazado entonces obtenemos la razón (`error`) y pedimos que nos devuelvan el dinero (`getRefund`).

```
const ticket = getFood();  
  
ticket  
  .then(food => eatFood(food))  
  .catch(error => getRefund(error));
```

Más info: Curso Open Webinars: Programación asíncrona con promises en JavaScript

[Mas información: https://platzi.com/blog/que-es-y-como-funcionan-las-promesas-en-javascript/](https://platzi.com/blog/que-es-y-como-funcionan-las-promesas-en-javascript/)

INTRODUCCIÓN



JavaScript



CÓMO FUNCIONA LA WEB



PETICIÓN A TRAVÉS DEL PROTOCOLO HTTP

`http://domain.com`



RESPUESTA (HTML, CSS, JS...)



Con AJAX interceptamos esta respuesta para evitar que la página se recargue. De esta forma en lugar de solicitar todos los datos (html,css,etc.), hacemos la petición por HTTP pero sólo solicitamos ciertos datos y los guardamos en un objeto.



PETICIÓN A TRAVÉS DEL PROTOCOLO HTTP

`http://domain.com`



{data}



RESPUESTA (data)



AJAX es asíncrono. Normalmente cuando hacemos una petición y esperamos una respuesta, el navegador se queda cargando y hasta que no recibimos una respuesta no se carga la página. Con AJAX conseguimos que esto no ocurra. Conseguimos que se cargue la página y que los datos solicitados lleguen mas tarde. Es decir, la página va por un lado y la petición y los datos van por otro.

Normalmente las peticiones a los servidores se suelen hacer con **PHP** o con **Node.js**



Para no tener que instalarnos php ni una base de datos para explicar la parte de AJAX vamos a utilizar una API que se llama JSONplaceholder. Podemos hacer peticiones normales o peticiones AJAX. <https://jsonplaceholder.typicode.com/>

Ejemplo con AJAX

```
const button = document.getElementById('button')

button.addEventListener('click', () => {
  let xhr
  if (window.XMLHttpRequest) xhr = new XMLHttpRequest()
  else xhr = new ActiveXObject("Microsoft.XMLHTTP")

  xhr.open('GET', 'https://jsonplaceholder.typicode.com/users')

  xhr.addEventListener('load', (data) => {
    console.log(data.target)
  })

  xhr.send()
})
```

Si solo se utiliza JQuery para peticiones AJAX podemos evitar usar JQuery con estas 2 líneas. XMLHttpRequest está presente a partir de Internet Explorer 11 debido a esto hay que controlar XMLHttpRequest está presente o no

Abrimos una petición al servidor

Hay que añadir un evento para que nos avise cuando se han cargado los datos debido a que estamos trabajando de forma asíncrona.

Enviamos la petición al servidor

Ejemplo con AJAX mostrando la información en un JSON

```
const button = document.getElementById('button')

button.addEventListener('click', () => {
  let xhr
  if (window.XMLHttpRequest) xhr = new XMLHttpRequest()
  else xhr = new ActiveXObject("Microsoft.XMLHTTP")

  xhr.open('GET', 'https://jsonplaceholder.typicode.com/users')

  xhr.addEventListener('load', (data) => {
    console.log(JSON.parse(data.target.response))
  })

  xhr.send()
})
```

Convertimos el String a JSON

EJERCICIO AJAX y JSON

- Modifica el ejemplo anterior de forma que muestre en mi página web en una lista, los datos del JSON recibidos. Deberá aparecer lo siguiente

AJAX

Get Data

- 1 - Leanne Graham
- 2 - Ervin Howell
- 3 - Clementine Bauch
- 4 - Patricia Lebsack
- 5 - Chelsey Dietrich
- 6 - Mrs. Dennis Schulist
- 7 - Kurtis Weissnat
- 8 - Nicholas Runolfsson
- 9 - Glenna Reichert
- 10 - Clementina DuBuque

FETCH API – MISMO EJEMPLO ANTERIOR

```
const button = document.getElementById('button')

//res = response = respuesta
button.addEventListener('click', () => {
  fetch('https://jsonplaceholder.typicode.com/users')
    .then(res => res.ok ? Promise.resolve(res) : Promise.reject(res))
    .then(res => res.json())
    .then(res => console.log(res))
})
```


FETCH API

- Es el reemplazo moderno del XMLHttpRequest
- Proporciona una interfaz JavaScript para acceder y manipular partes del canal HTTP, como peticiones y respuestas.
- También provee un método global `fetch()` que proporciona una forma fácil y lógica de obtener recursos de forma asíncrona por la red. Está basado en promesas, por lo cual tiene un `response` y un `reject` internos. `Response` tiene varios métodos
 - **`arrayBuffer()`**: Archivos binarios en bruto (mp3, pdf, jpg, etc). Se utiliza cuando se necesita manipular el contenido del archivo.
 - **`blob()`**: Archivos binarios en bruto (mp3, pdf, jpg, etc). Se utiliza cuando no se necesita manipular el contenido y se va a trabajar con el archivo directamente
 - **`clone()`**: crea un clon de un objeto de respuesta, idéntico en todos los sentidos, pero almacenado en una variable diferente.
 - **`formData()`**: Se utiliza para leer los objetos `formData`
 - **`json()`**: Convierte los archivos json en un objeto de JavaScript
 - **`text()`**: Se utiliza cuando queremos leer un archivo de texto. Siempre se codifica en UTF-8
- En Internet Explorer no funciona

EJERCICIO AJAX y JSON

- Modifica el ejemplo anterior de forma que muestre en mi página web en una lista, los datos del JSON recibidos utilizando **FETCH**. Deberá aparecer lo siguiente

AJAX

Get Data

- 1 - Leanne Graham
- 2 - Ervin Howell
- 3 - Clementine Bauch
- 4 - Patricia Lebsack
- 5 - Chelsey Dietrich
- 6 - Mrs. Dennis Schulist
- 7 - Kurtis Weissnat
- 8 - Nicholas Runolfsson
- 9 - Glenna Reichert
- 10 - Clementina DuBuque

FETCH API

- Es el reemplazo moderno del XMLHttpRequest
- Proporciona una interfaz JavaScript para acceder y manipular partes del canal HTTP, como peticiones y respuestas.
- También provee un método global `fetch()` que proporciona una forma fácil y lógica de obtener recursos de forma asíncrona por la red. Está basado en promesas, por lo cual tiene un `response` y un `reject` internos. `Response` tiene varios métodos
 - **`arrayBuffer()`**: Archivos binarios en bruto (mp3, pdf, jpg, etc). Se utiliza cuando se necesita manipular el contenido del archivo.
 - **`blob()`**: Archivos binarios en bruto (mp3, pdf, jpg, etc). Se utiliza cuando no se necesita manipular el contenido y se va a trabajar con el archivo directamente
 - **`clone()`**: crea un clon de un objeto de respuesta, idéntico en todos los sentidos, pero almacenado en una variable diferente.
 - **`formData()`**: Se utiliza para leer los objetos `formData`
 - **`json()`**: Convierte los archivos json en un objeto de JavaScript
 - **`text()`**: Se utiliza cuando queremos leer un archivo de texto. Siempre se codifica en UTF-8
- En Internet Explorer no funciona

FETCH API -POST

- Para hacer peticiones POST, fetch admite un segundo parámetro.

```
fetch(url, {  
  method: 'POST',  
  body: Los datos que enviamos. Si es un objeto hay que convertirlo con  
  JSON.stringify(datos),  
  headers: {          cabeceras de información sobre lo que estamos enviando  
    https://developer.mozilla.org/es/docs/Web/HTTP/Headers    }  
})
```

FETCH API –POST. EJEMPLO

```
const button = document.getElementById('button')

button.addEventListener('click', () => {
  const newPost = {
    title: 'A new post',
    body: ' Lorem ipsum dolor sit amet consectetur adipisicing
elit.',
    userId: 1
  }

  fetch('https://jsonplaceholder.typicode.com/posts', {
    method: 'POST',
    body: JSON.stringify(newPost),
    headers: {
      "Content-type": "application/json"
    }
  })
    .then(res => res.json())
    .then(data => console.log(data))
})
```


EJERCICIO AJAX, JSON Y BASE DE DATOS

1. Crea la base de datos que se proporciona en el archivo marvel.sql
2. Interpreta el código marvel.php y colócalo en el servidor y utilízalo cuando creas conveniente.
3. Crea una página html que muestre:
 - Un tabla cuyos títulos sean: Name, Gender, Fighting Skills
 - Un campo de tipo select donde deberán mostrarse los superheroes que hay cargados en la base de datos.
4. Crea un botón de tipo submit que se llame "Get Data"
5. Al pulsar el botón se deberá modificar el comportamiento para que en lugar de realizar el submit por defecto se llame una nueva función llamada "getData".
6. Crea una función que se llame getData y que reciba un id. Dentro de esta función , se deberá realizar una petición AJAX que recupere los datos de la base de datos en un JSON y que rellene el select creado de superheroes si este no está relleno. En caso de estar relleno, al seleccionar un superheroe del select y pulsar el botón "Get Data" deberá crear una fila en la tabla con las características del superheroe seleccionado.

EJERCICIO AJAX, LECTURA DE ARCHIVOS

1. Crea una página con 2 botones. Uno para mostrar una imagen y otro para mostrar un pdf
2. Utilizando fetch. Al pulsar el botón de mostrar imagen deberá mostrar la imagen. Para ello puedes usar la función `blob()` y `URL.createObjectURL`. Busca cómo funcionan
3. Utilizando fetch. Al pulsar el botón mostrar pdf deberá mostrar el pdf a través de un link.

API –WEB STORAGE

Nos permite guardar información en el dispositivo con el conjunto de clave – valor. Muy parecido a una cookie o a un objeto Javascript pero el tamaño es mucho mas grande.

Tiene dos mecanismos en el almacenamiento web que son los siguientes:

- `sessionStorage` mantiene un área de almacenamiento separada para cada origen que está disponible mientras dure la sesión de la página (mientras el navegador esté abierto, incluyendo recargas de página).
- `localStorage` hace lo mismo, pero persiste incluso cuando el navegador se cierre y se reabra.

Ambos funcionan con `clave:valor` y tienen dos métodos fundamentales:

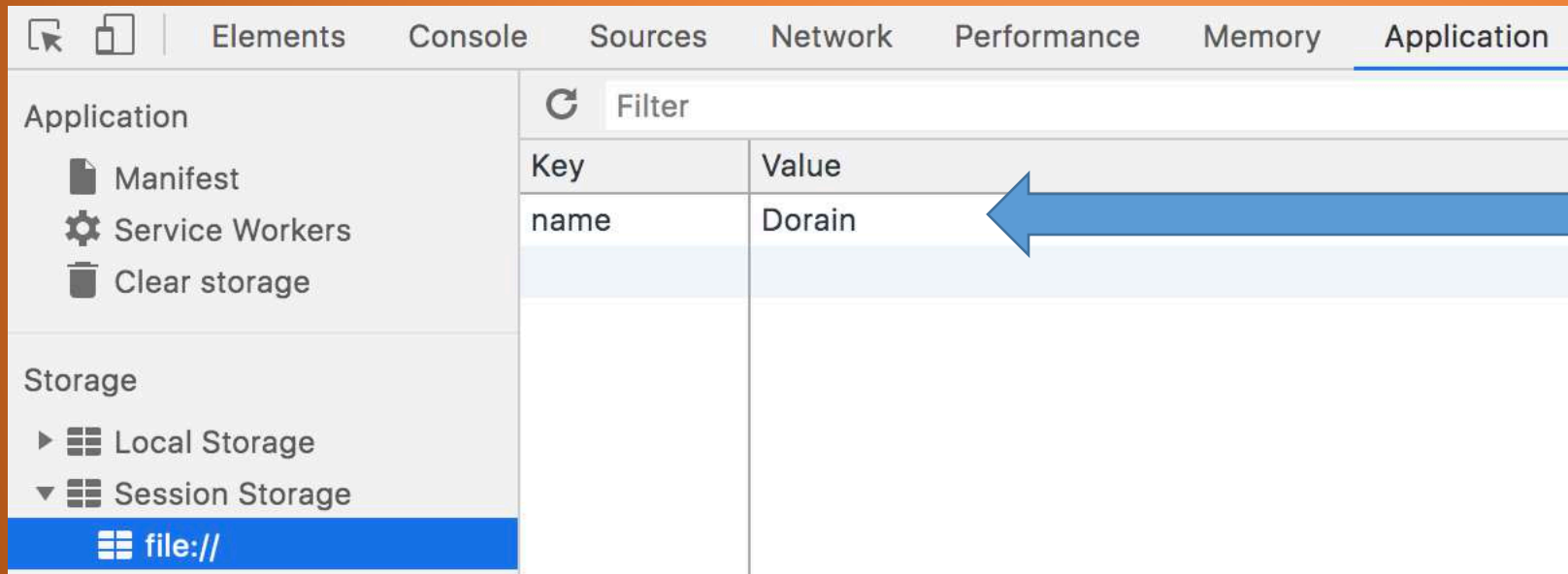
- `setItem()` para asignar una clave:valor
- `getItem()` que recibe como parámetro la clave de la que queremos obtener el valor

API –WEB STORAGE. EJEMPLO

```
const form = document.getElementById('form')
const keys = document.getElementById('keys')

form.addEventListener('submit', (e) => {
  e.preventDefault()

  sessionStorage.setItem('name', 'Dorian')
})
```



The screenshot shows the Chrome DevTools 'Application' tab. On the left sidebar, under 'Storage', 'Session Storage' is expanded, and 'file://' is selected. The main panel displays a table of stored items:

Key	Value
name	Dorain

A blue arrow points from the text on the right to the 'Dorain' value in the table.

Si pulsamos F12 en nuestro navegador podemos ver los valores del Web Storage

API –WEB STORAGE. EJEMPLO CON OBJETO JAVASCRIPT

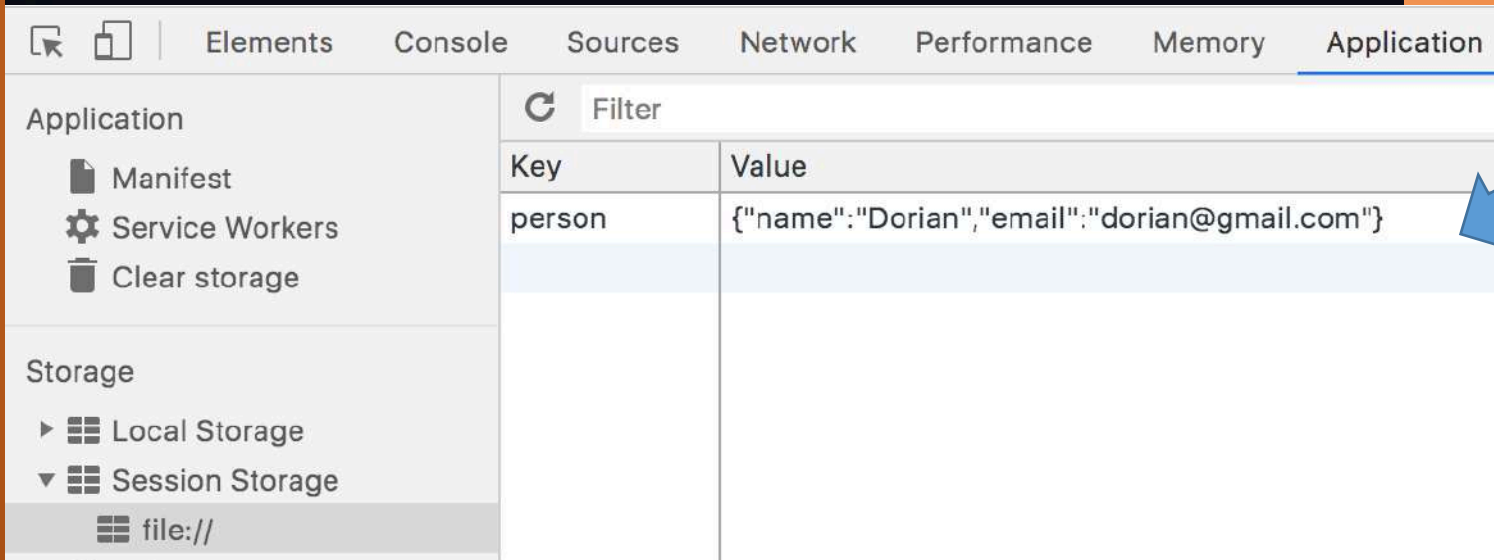
```
const form = document.getElementById('form')
const keys = document.getElementById('keys')

form.addEventListener('submit', (e) => {
  e.preventDefault()

  const person = {
    name: 'Dorian',
    email: 'dorian@gmail.com'
  }

  sessionStorage.setItem('person', JSON.stringify(person))
})
```

Como tiene que recibir una cadena, convertimos el objeto a cadena



The screenshot shows the Chrome DevTools Application tab. On the left, under 'Storage', 'Session Storage' is expanded, showing 'file://'. The main area displays a table of session storage items. A blue arrow points from the text 'Como tiene que recibir una cadena, convertimos el objeto a cadena' to the 'person' entry in the table.

Key	Value
person	{"name":"Dorian","email":"dorian@gmail.com"}

API –WEB STORAGE. EJEMPLO CON DATOS DE FORMULARIO

Key

Value

Select key

```
sessionStorage.setItem(form.key.value, form.value.value)
```

Dado el siguiente formulario, suponiendo que lo tenemos cargado en la variable `form`. Podemos guardar el valor del input `key` y el valor del input `value` en el `sessionStorage` de esta manera

Para recuperar valores del `sessionStorage` podemos utilizar la siguiente instrucción:
`sessionStorage.getItem(nombre de la clave a recuperar)`

API –WEB STORAGE. EJEMPLO CON DATOS DE FORMULARIO

- Para recuperar valores del `sessionStorage` podemos utilizar la siguiente instrucción: `sessionStorage.getItem(nombre de la clave a recuperar)`
- Para limpiar los datos utilizaríamos `sessionStorage.clear()`
- Para borrar un elemento utilizaríamos `sessionStorage.removeItem(clave de lo que queremos borrar)`

TODOS LOS MÉTODOS DE `sessionStorage` ME SIRVEN DE IGUAL MANERA PARA `local Storage`