UT 8 - EVENTOS EN LA UI (RESUMEN)

Contenido

1. Eventos del Mouse

Eventos al mover el interactuar con un dispositivo de tipo puntero (como un ratón).

Evento	Se produce cuando
`mouseover`	El puntero del mouse se mueve hacia el elemento. Se dispara también si el mouse se mueve hacia uno de sus descendientes.
`mouseout`	El puntero del mouse se mueve hacia fuera del elemento. Se dispara también cuando el mouse se mueve hacia fuera de uno de sus descendientes. Se dispara también cuando el mouse entra en uno de sus descendientes.
`mousemove`	Se produce cualquier movimiento del mouse sobre un elemento.

Eventos al hacer clic con un botón del ratón

Evento	Se produce cuando
`mousedown` `mouseup`	Se oprime/suelta el botón del ratón sobre un elemento.
`click`	Se activa al pulsar el botón izquierdo del ratón. Este evento se produce después del `mousedown` y el `mouseup` sobre un elemento.
`dblclick`	Se activa al pulsar dos veces seguidas el botón primario (*izquierdo*) del ratón. Este evento se produce después de dos eventos click (con sus correspondientes mousedown y mouseup). Hoy apenas se usa.
`contextmenu`	Se activa al pulsar el botón secundario (*derecho*) del ratón. Existen otras formas de abrir el menú contextual, por ejemplo: usando un comando especial de teclado también puede activarse, de manera que no es exactamente un evento exclusivo del mouse.

El orden de los eventos al hacer clic con un botón del ratón

`mousedown` → `mouseup` → `click` → `mousedown` → `mouseup` → `click` → dbl`click`

El botón del mouse

Propiedades relacionadas con los eventos `mousedown` y `mouseup`.

Propiedad	Descripción / Valor	
`button`	Contiene un número entero que indica qué botón se ha pulsado.	
	0	Botón izquierdo (primario-*derecho*)
	1	Botón central (auxiliar)
	2	Botón derecho (secundario-*izquierdo*)
	3	Botón X1 (atrás)
	4	Botón X2 (adelante)

`witch`	Forma antigua y no estándar de obtener el botón pulsado. Propiedad obsoleta (*deprecated*).
`buttons`	Entero que guarda todos los botones presionados (un bit por evento). Propiedad poco utilizada

Modificadores: shift, alt, ctrl y meta

Todos los eventos del ratón incluyen información sobre las teclas modificadoras presionadas.

Para ello, existen unas propiedades de tipo boolean: son true si la tecla fue presionada durante el evento.

Propiedad	Tecla presionada
`shiftKey`	Mayúsculas (`Shift`)
`altKey`	`Alt` (o `Opt` para Mac)
`ctrlKey`	`Ctrl`
`metaKey`	`Cmd` para Mac

Nota

En la mayoría de las aplicaciones, cuando Windows/Linux usan `Ctrl`, en Mac se usa `Cmd`.

Por tanto, si queremos comprobar si se ha pulsado la tecla `Ctrl`, y dar soporte a todos los sistemas operativos, tendremos que comprobar:

```
if (event.ctrlKey || event.metaKey).
```

Coordenadas

Propiedades del evento que proporcionan las coordenadas del ratón

Propiedad	Tecla presionada
`event.clientX` `event.clientY`	Coordenadas relativas a la ventana.
`event.pageX` `event.pageY`	Coordenadas relativas al documento HTML.

Impedir la selección

Al hacer doble clic sobre un texto, ese texto se selecciona.

Si se hace clic sobre un texto y se arrastra, ese texto también se selecciona.

Para evitarlo, se puede impedir el comportamiento por defecto del evento `mousedown.`

Ejemplo

```
 <p ondblclick="alert('Click!')" onmousedown="return
false">texto</p>
```

Impedir que se copie al portapapeles un texto seleccionado

Para ello se impide el comportamiento por defecto del evento `oncopy`.

Ejemplo

```
<div oncopy="alert('¡Copiado prohibido!');return false">
  texto</div>
```

2. Moviendo el mouse: mouseover/out, mouseenter/leave

Eventos al mover un dispositivo apuntador (como un ratón)

Eventos mouseover/mouseout/ relatedTarget

Evento	Se produce cuando
`mouseover`	El puntero del mouse se mueve hacia el elemento. Se dispara también si el mouse se mueve hacia uno de sus descendientes.
`mouseout`	El puntero del mouse se mueve hacia fuera del elemento. Se dispara también cuando el mouse se mueve hacia fuera de uno de sus descendientes. Se dispara también cuando el mouse entra en uno de sus descendientes.
`mousemove`	Se produce cualquier movimiento del mouse sobre un elemento.

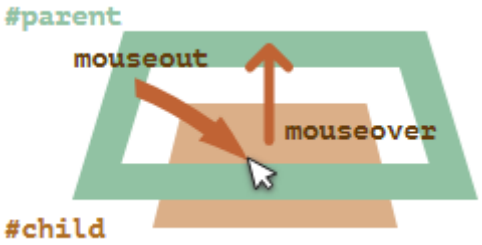
* Si el movimiento es muy rápido, es posible que se salten elementos, y los eventos sólo se produzcan sobre algunos de ellos.

Propiedades del evento relacionadas

Propiedad	Descripción
`ev.relatedTarget`	Elemento hacia/desde el que se mueve el ratón. Puede ser `null`, si el ratón se mueve desde/hacia fuera del documento.

Mover hacia/desde un hijo

* Si el ratón se mueve hacia uno de los hijos, se produce el `mouseout` en el padre, y el `mouseover` en el hijo. Y, al moverse hacia el padre, se produce el `mouseout` en el hijo y el `mouseover` en el padre.



Eventos mouseenter/mouseleave

Eventos similares a mouseover/mouseout.

Diferencias:

1. Las transiciones hacia/desde los descendientes no se cuentan.
2. Los eventos mouseenter/mouseleave no "burbujean", no se pueden detectar en sus contenedores padre. No se puede utilizar la delegación de eventos con ellos.

Evento	Se produce cuando
`mouseenter`	El puntero del mouse se mueve hacia el elemento.
`mouseleave`	El puntero del mouse se mueve hacia fuera del elemento.

~~3. Arrastrar y soltar con eventos del ratón~~

Este apartado no lo veremos en clase.

En el estándar de HTML existen eventos como "dragstart y dragend" para gestionar las operaciones de arrastrar y soltar. Sin embargo, esos eventos tienen limitaciones. En este apartado se explica cómo implementar "Arrastrar y Soltar" utilizando eventos del ratón.

~~4. Eventos de puntero~~

Este apartado no lo veremos en clase.

Cuando aparecieron los dispositivos táctiles, los eventos de ratón se disparaban cuando se producía un movimiento con un dispositivo señalador. Sin embargo, en los dispositivos táctiles se pueden realizar más operaciones que en un dispositivo dónde sólo puedes interactuar con un ratón.

Para resolver estos problemas, se introdujo un nuevo estándar, que proporciona un conjunto único de eventos para todo tipo de eventos señaladores.

5. Teclado: keydown y keyup

Estos eventos se pueden utilizar cuando se quiere gestionar acciones de teclado (también pueden ser teclados virtuales). Por ejemplo, reaccionar cuando se presionan las teclas Arriba, Abajo, teclas de función, o incluso combinación de teclas.

Evento	Se produce cuando
`keydown`	Cuando se presiona una tecla
`keyup`	Cuando se libera una tecla

Propiedades asociadas con los eventos keydown y keyup

Propiedad	Descripción / Valor	
`event.key`	Carácter que se ha obtenido (ej. "z" o "Z")	
`event.code`	Código de la tecla pulsada (ej. "**`KeyZ`**")	
	`"Key<letter>"`	Para las teclas de las letras. Ej. "**`KeyA`**", "**`KeyB`**"...
	`"Digit<number>"`	Para las teclas de los números. Ej . "**`Digit0`**", "**`Digit1`**", ...
		Las teclas especiales tienen sus propios nombres. Ej. "**`Enter`**", "**`Backspace`**", "Tab", ...
`event.repeat`	Si una pulsada se ha mantenido presionada durante un tiempo "largo".	

Se pueden consultar todos los valores de las propiedades `event.key` o `event.code` en este enlace: https://www.w3.org/TR/uievents-code/.

Auto-repeat

Si se mantiene una tecla presionada durante un tiempo "largo", el evento `keydown` se dispara muchas veces, y cuando se suelta, se dispara un evento `keyup`.

Acciones por defecto

Las acciones por defecto son variadas:

- Si se presiona un carácter, aparece en pantalla.
- Si se presiona la tecla `Delete` , se borra un carácter.
- Etc.

El comportamiento por defecto del evento `keydown` se puede cancelar, con algunas excepciones (eventos de teclado dirigidos al sistema operativo). Por ejemplo, si se presiona `Alt+F4` se cierra la ventana actual del navegador. Y no hay modo de evitar este comportamiento con JavaScript.

Eventos y propiedades DEPRECATED

El evento `keypress`, y las propiedades `keyCode`, `charCode` y `which` no deben utilizarse ya.

~~6. Desplazamiento~~

Este apartado no lo veremos en clase.

En este apartado se ve el evento scroll que permite reaccionar al desplazamiento de una página o elemento.

UT 9 - FORMULARIOS Y CONTROLES (RESUMEN)

Propiedades y eventos especiales para formularios (<form>) y controles (<input>, <select> y otros).

Contenido

1. Propiedades y métodos de formularios

Navegación: formularios y elementos

Los formularios del documento son miembros de una colección especial `document.forms`.

También se llama "colección nombrada": se puede acceder por el nombre del formulario (propiedad `name`) y por un índice.

Propiedad	Descripción
`document.forms`	Colección nombrada de formularios del documento
`document.forms.name` *`formulario`*`.elements["name"]`	Puntero al formulario con ese "name"
`document.forms[0]`	Puntero al primer formulario del documento

Para cada formulario, se crea una "colección nombrada" de sus controles.

Si hay varios elementos con el mismo name, en lu

Propiedad	Descripción
`formulario``.elements`	Colección nombrada de controles del formulario
`formulario``.elements.name` *`formulario`*`.elements["name"]`	Puntero al control del formulario con ese "name"(*)
`formulario``.elements[0]`	Puntero al primer control del formulario (*)

(*) Si hay varios elementos con el mismo `name` (`checkbox`, `radio`), cada elemento de la colección `elements`, será a su vez una colección.

Fieldset como "subformularios"

Los elementos `<fieldset>` son como sub-formularios, ya que tienen una propiedad `elements` con todos los elementos que están dentro del formulario.

Notación corta: *formulario*.name

En lugar de *`formulario`*`.elements.name`, podemos escribir *`formulario`*`.name`.

Propiedad	Descripción
`formulario``.name` *`formulario`*`["name"]`	Puntero al control del formulario con ese "name"(*)

Referencia inversa: element.form

Todos los controles de un formulario tienen una propiedad `form`, que apunta al formulario.

Propiedad	Descripción
`control``.form`	Puntero al formulario al que pertenece ese control.

Elementos del formulario (controles)

<input> y <textarea>

Propiedad	Descripción
`input.value`	Cadena de texto con la información del `input`
`input.checked`	Boolean que indica si el `input` está seleccionado o no. Para `<input type="checkbox">` o `<input type="radio">`
`textarea.value`	Cadena de texto con la información del `textarea`. (*) No utilizar `innerHTML`, porque guarda el HTML que había inicialmente en la página, no su valor actual

<select> y <option>

Propiedad	Descripción
`select.options`	Colección de subelementos `<option>`
`select.value`	Valor del `<option>` seleccionado actualmente
`select.selectedIndex`	Número del `<option>` seleccionado actualmente.
`option.selected`	Boolean que indica si un `<option>` está seleccionado.
`option.index`	Número del `option` respecto a los demás en su `select`.
`option.text`	Contenido del `option`.

Cómo asignar un valor a un select: tres métodos

- select.value = value del option que queremos seleccionar
- select.selectedIndex = índice del option que queremos seleccionar
- option.selected = true

(*) Si el `<select>` tiene activado el atributo multiple, permite seleccionar varios `<option>`.
En ese caso, habrá que utilizar el último método.

new Option

Permite crear fácilmente elementos <option>.

```
option = new Option(text, value, defaultSelected, selected)
```

Ejemplo:

```
let option = new Option("Text", "value");

// crea <option value="value">Text</option>
```

2. Foco: focus / blur

Un elemento se puede "enfocar" de varios modos: haciendo click sobre el con Tab, etc.

Cuando un elemento se enfoca, la entrada desde teclado (o desde un paste) irá dirigida a ese elemento.

Cuando un elemento pierde el foco (se "desenfoca", "blur"), la entrada desde teclado ya no irá dirigida a ese elemento. Pero además, indica que los datos ya han sido introducidos. Puede ser el momento de ejecutar código que valide los datos.

Eventos focus y blur

Evento	Descripción
`focus`	Un elemento obtiene el foco. Este evento no se propaga.
`blur`	Un elemento pierde el foco. Este evento no se propaga

Métodos focus y blur

Método	Descripción
`focus()`	Pone el foco sobre el elemento
`blur()`	Quita el foco del elemento

(*) El foco se puede perder cuando el usuario hace clic en otro lado. Pero el propio JavaScript puede causarlo:

- Un `alert` traslada el foco hacia sí mismo. El elemento que lo tenía, lo pierde.
- Si un elemento se elimina, pierde el foco.

~~Permitir enfocado sobre cualquier elemento~~

Por defecto, no todos los elementos permiten el enfoque. Lo permiten aquellos con los que el visitante puede interactuar `<button>`, `<input>`, `<select>`, `<a>`, etc.

Pero este comportamiento por defecto se puede modificar. Si un elemento html tiene el atributo `tabIndex`, puede tener el foco. Los valores de este atributo son 1, 2, 3... indican que un elemento puede tener el foco y en qué orden se asigna. Si un elemento tiene `tabIndex`=0, puede tener el foco, pero obtiene el foco en el orden que aparece en el código.

Se puede asignar a un elemento el atributo `tabIndex` con `elemento.setAttribute("tabIndex")` o con `elemento.tabIndex=.`

~~Delegación: focusin/focusout~~

Los eventos `focus` y `blur` no se propagan, por lo que no se puede utilizar delegación de eventos. Existen dos eventos `focusin` y `focusout` que son equivalentes a `focus` y `blur`, pero se propagan.

Evento	Descripción
`focusin`	Un elemento obtiene el foco. Este evento se propaga.
`focusout`	Un elemento pierde el foco. Este evento se propaga.

3. Eventos: change, input, cut, copy, paste

Evento: change

Se activa cuando el elemento finaliza un cambio.

- input de tipo texto: cuando pierde el foco.
- input de tipo `boolean` (`type`=`checkbox`/`radio`): cuando se selecciona una opción.

Evento:input

Se dispara cada vez que un valor es modificado por el usuario. No espera a que finalice el cambio. Ejemplo, cuando escribimos o borramos una letra dentro del `input`.

Eventos:cut, copy, paste

Se disparan al cortar/copiar/pegar un valor.

Se puede acceder a los datos cortados/copiados/pegados.

Se puede impedir su funcionamiento con `event`.`preventDefault`().

Método	Descripción
`event.clipboardDate(getData("text/plain"))`	Contenido del portapapeles
`document.getSelection()`	Contenido del portapapeles

4. Formularios: evento y método submit

Evento: submit

Es un evento del formulario.

Se activa cuando el formulario es enviado, normalmente haciendo clic en un botón de `type=submit`, o pulsando `enter` en un campo de texto del formulario.

Se suele utilizar para validar el formulario y permitir el envío al servidor, o impedirlo.

Método: submit()

Es un método del formulario.

Permite enviar el formulario al servidor manualmente. Si se llama a este método, no se dispara en evento `submit`.

UNIDAD 9 (2). Formularios - Expresiones regulares

Desarrollo Web en Entorno Cliente

2º DAW

Contenidos

Expresiones regulares

Las expresiones regulares son un elemento de uso muy habitual en la mayoría de lenguajes de programación. Se utilizan principalmente, para definir patrones que reconocen cadenas de caracteres específicas. Estas condiciones nos facilitan la tarea de búsqueda de textos dentro de otros textos, validación de formularios o extracción avanzada de subcadenas.

Las expresiones regulares de JavaScript son objetos de tipo **`RegExp`** que se pueden crear delimitándolas entre dos barras (/) al principio y al final. Tras las barras se pueden indicar controles o banderas para crear expresiones regulares avanzadas. También se pueden crear utilizando su constructor. La sintaxis sería la siguiente:

```
let expReg = /expresión/[controles];
let expReg = new RegExp("expresión", [controles]);
```

Una expresión regular se puede crear a través de su literal o. Ejemplo:

```
let expReg1 = /\w+/;
let expReg2 = new RegExp("\\w+");
```

Métodos para trabajar con expresiones regulares: SOLO TEST

Las expresiones regulares se utilizan fundamentalmente con dos métodos de la clase **`String`**:

Método	Devuelve	Descripción
`cadena.match(/expReg/)`	`[String]`	Devuelve un array con primer substring que casa o `null`. Algunos navegadores devuelven un array con más parámetros, pero conviene utilizar solo el primero.
`cadena.replace(/expReg/, repuesto)`	`String`	Devuelve un string resultado de sustituir el primer substring que casa por *repuesto*.

Ejemplos:

```
"Es hoy".match(/hoy/) => devuelve ["hoy"]
"Número: 142719".replace(/1/, "x") => devuelve "Número: x42719"
```

Cuando se validan campos de formularios, es muy frecuente tener que comprobar si una cadena tiene un formato determinado. Esto se puede hacer comprobando si una cadena casa con una expresión regular utilizando el siguiente método de la clase **`RegExp`**:

Método	Devuelve	Descripción
`expReg.test(cadena)`	`Boolean`	Devuelve `true` si la cadena casa con la expresión regular, `false` en otro caso.
`exReg.exec(cadena)`	`[String]`	Devuelve un array con primer substring que casa o `null`. Se comporta como `match()`.

Ejemplo:

```
let expReg = /[0-9]{8}[a-zA-Z]/
expReg.test("012345678A")   => devuelve true
```

Construcción de expresiones regulares

Algunos patrones básicos

Carácter	Significado
c	Carácter. Siendo c un carácter cualquiera, encaja solamente con ese carácter.
cde	Secuencia. Siendo c, d, y e tres caracteres cualquiera, encaja solamente si esos caracteres aparecen de esa manera. Ejemplo: `/hola/` encaja con textos que contengan la palabra hola
^	Principio de cadena. Ejemplo: `/^A/` encaja con la "A" en "Asturias" pero no encaja con la "A" en "Es Asturias".
$	Final de cadena. Ejemplo: `/s$/` encaja con última "s" en "Asturias" pero no con la primera (segunda letra).
.	Cualquier carácter excepto fin de línea.

Ejemplos:

```
"Es hoy".match(/hoy$/) => devuelve ['hoy'] (está al final)
"Es hoy".match(/^hoy/) => devuelve null (no está al principio)
"Es hoy".match(/^..../) => devuelve los 4 primeros caracteres:
['Es h']
```

Clases y rangos de caracteres

Carácter	Significado
[xyz]	Clase de caracteres. Encaja con uno de los caracteres que están entre los corchetes. Ejemplos: `/chic[ao]/` cadenas `"chico"` y `"chica"`. `/[aeiou]/` cualquier vocal (minúscula).
[^xyz]	Clase de caracteres negada. Encaja con un carácter que NO esté entre los corchetes. Ejemplo: `/[^aeiou]/` no debe ser vocal (minúscula).
[a-z]	Rango de caracteres. En este corresponde al rango “a-z” de letras ASCII. Ejemplo: `/[0-9][0-9][0-9]/` las cadenas que contienen tres números seguidos, como `"123"` y `"676"` encajan con el patrón.
[^a-z]	Rango de caracteres negado. Ejemplo: `/[^c-z]a/` las cadenas `"aa"` y `"ba"` encajan, pero la cadena `"za"` no encaja.

Ejemplos:

```
"canciones".match(/[aeiou]/) => devuelve ['a']
"canciones".match(/c[aeiou]/) => devuelve ['ca']
"canciones".match(/n[aeiou]/) => devuelve ['ne']
```

Otras clases y rangos de caracteres

Carácter	Significado
\d	Cualquier dígito. Equivale a `[0-9]`. Ejemplo: `/\d/` o `/[0-9]/` encaja con "2" en "B2".
\D	Cualquier carácter que no sea un dígito. Equivale a `[^0-9]`. Ejemplo: `/\D/` o `/[^0-9]` encaja con "B" en "B2".
\w	Cualquier carácter alfanumérico, incluido el subrayado. Equivale a [A-Za-z0-9_]. Ejemplo: `/\w/` encaja con "a" en "apple", "5" en "$5.28" y "3" in "3D".
\W	Cualquier carácter que no es un carácter alfanumérico o subrayado. Equivale a [^A-Za-z0-9_] Ejemplo: `/\W/` encaja con "." y con "$" en "$5.28".
\s	Cualquier carácter de tipo espacio en blanco, como espacio en blanco, tabulador, salto de línea. Reconoce separadores [\f\n\r\t\v\u00a0\u1680 .....]. Ejemplo: `/\s\w*/` encaja con " bar" en "foo bar".
\S	Cualquier carácter que no sea de tipo espacio en blanco.
\t	Tabulador horizontal.
\r	Salto de línea.
\n	Nueva línea.
\v	Tabulador vertical.
\f	Salto de página.
****	Para escapar un carácter que tiene un significado especial, como [,], /, \, etc. Por ejemplo \\ es la forma de representar la barra invertida.
\uffff	Permite indicar un carácter Unicode mediante su código hexadecimal.
\p{PropUnicode}	Cualquier carácter que pertenezca a la propiedad Unicode indicada.
\P{PropUnicode}	Cualquier carácter que no pertenezca a la propiedad Unicode.

Controles i, g, m

Carácter	Significado
i	Búsqueda insensible a mayúsculas.
g	Búsqueda global de todos los substrings que casan con el patrón (no solamente con el primero).
m	Búsqueda multilínea, donde ^ y $ representan principio y fin de línea.

Ejemplos con match():

```
"canciones".match(/[aeiou]/g) => devuelve ['a', 'i', 'o', 'e']
"canciones".match(/c[aeiou]/g) => devuelve ['ca', 'ci']
"Hoy dice hola".match(/ho/i) => devuelve ['Ho']
"Hoy dice hola".match(/ho/ig) => devuelve ['Ho', 'ho']
"Hola Pepe\nHoy vás".match(/^Ho/g) => devuelve ['Ho']
"Hola Pepe\nHoy vás".match(/^ho/gim) => devuelve ['Ho', 'Ho']
```

Ejemplos con replace():

```
"Número: 142719".replace(/1/, 'x') => devuelve 'Número: x42719'
"Número: 142719".replace(/1/g, 'x') => devuelve 'Número: x427x9'
```

```
"Número: 142719".replace(/[0-9]+/, '<número>') => devuelve
'Número: <número>'
```

Operadores de repetición

Carácter	Significado
*	Cero o más veces. Ejemplo: `/a*/` encaja con: “”, “a”, “aa”, “aaa”, ...
+	Una o más veces. Ejemplo: `/a+/` encaja con: “a”,“aa”, “aaa”, …
?	Cero o una vez. Ejemplo: `/a?/` encaja solo con: “” y “a”
{n}	n veces. Ejemplo: `/a{2}/` encaja solo con: “aa”
{n,}	n o más veces. Ejemplo: `/a{2,}/` encaja con: “aa”, “aaa”, “aaaa”, ...
{n, m}	Entre n y m veces. Ejemplo: `/a{2,3}/` encaja solo con: “aa” y “aaa”

Ejemplos:

```
"tiene".match(/[aeiou]+/g) => devuelve ['ie', 'e']
// cadenas no vacías de vocales
"tiene".match(/[aeiou]?/g) => devuelve ['', 'i', 'e', '', 'e', '']
// vocal o nada
"tiene".match(/[aeiou]*/g) => devuelve ['','ie', '', 'e', '']
// cadenas de vocales incluyendo ""
"Había un niño.".match(/[a-zñáéíóú]+/ig) => devuelve ['Había',
'un', 'niño']
// palabras en castellano: ASCII extendido con ñ, á, é, í, ó, ú
```

Los operadores de repetición son “ansiosos” y siempre buscarán encajar o casar con la cadena más larga posible, pero pueden volverse “perezosos” añadiendo `?` detrás del operador de repetición, en ese caso casarán con la cadena más corta posible.

Ejemplos:

```
"aaabb".match(/a+/) => devuelve ['aaa']
"aaabb".match(/a+?/) => devuelve ['a']
"ccaaccbccaa".match(/.+cc/) => devuelve ['ccaaccbcc']
"ccaaccbccaa".match(/.+?cc/) => ['ccaacc']
```

Estos dos patrones se utilizan mucho:

- `.+` cualquier cadena con longitud uno o más
- `.*` cualquier cadena con longitud cero o más

Patrones alternativos

Carácter	Significado

x|y	Encaja con x o con y. Ejemplo: `/green|red/` encaja con "green" en "green apple" y con "red" in "red apple".

Ejemplos:

```
"canciones".match(/ci|ca/) => devuelve ['ca']
"canciones".match(/ci|ca/g) => devuelve ['ca','ci']
"1 + 2 --> tres".match(/[a-z]+|[0-9]+/g) => devuelve ['1','2','tres']
```

~~Subpatrones~~

Carácter	Significado
(x)	Subpatrón. Encaja con x y recuerda la cadena encajada. El patrón y los subpatrones pueden reutilizarse referenciándolos con: o $0 representa todo el patrón o $1 representa el primer subpatrón o $2 el match el segundo subpatrón y así sucesivamente Ejemplos: `/(c)([aeiou])/` patrón c seguido de vocal, delimitando la c como primer subpatrón ($1) y las vocales como segundo subpatrón ($2) `/([0-9]+)(,[0-9]*)?/` patrón que representa números enteros o decimales, tiene dos subpatrones `([0-9]+)` y `(,[0-9]*)`

El método `match()` realmente busca patrones y subpatrones, por lo tanto, `cadena.match(/patrón/)` devolverá el match completo seguido de los subpatrones: `[match$0, macth$1, match$2,…]`

Ejemplos con match():

```
"canciones".match(/(c)([aeiou])/) => devuelve ['ca','c','a']
"canciones".match(/c([aeiou])n/) => devuelve ['can','a']
"canciones".match(/(..)..(..)/) => devuelve ['cancio','ca','io']
```

Ejemplos con replace():

```
//Sustituye por el primer subpatrón
"Número: 142,719".replace(/([0-9]+)(,[0-9]*)?/, '$1') => devuelve
'Número: 142'
//Sustituye por 0 y segundo supbatrón
"Número: 142,719".replace(/([0-9]+)(,[0-9]*)?/, '0$2') =>
devuelve 'Número: 0,719'
//Sacamos la , del subpatrón para que la sustituya por el .
"Número: 142,719".replace(/([0-9]+),([0-9]*)?/, '$1.$2') =>
devuelve 'Número: 142.719'
```

Ejemplo con test() y exec():

```
//Patrón que representa un código postal (formado por 5 números
del 00000 al 52999)
let cp = /^((5[012])|([0-4][0-9]))([0-9]{3})$/;
cp.test("49345") => devuelve true
```

```
cp.test("53345") => devuelve false
cp.exec("49345") => devuelve ['49345']
cp.exec("53345") => devuelve null
```

UNIDAD 9 (3) - Formularios - Almacenar datos en el equipo cliente

Desarrollo Web en Entorno Cliente

2º DAW

Contenidos

Almacenar datos en el equipo cliente

Cookies

Con `document.cookie` se obtienen y establecen las cookies asociadas al documento.
Las cookies son datos almacenados en nuestro ordenador, en pequeños archivos de texto.
Van ligadas a los navegadores y a los documentos: las que vemos en un navegador, no se pueden acceder desde otro.
Pueden tener fecha de expiración. Si no la tienen, expiran al acabar la sesión.

Leer todas las cookies accesibles

```
let todasCookies = document.cookie;
```

En el código anterior, `todasCookies` tendrá una cadena formada por una lista de todas las cookies (con formato clave=valor) separadas por punto y coma.

Ejemplo: Esta es la cadena obtenida al ejecutar el código anterior con https://devdocs.io:

```
"_gauges_unique_month=1; _gauges_unique_year=1;
docs=css/dom/dom_events/html/http/javascript; schema=2;
_ga=GA1.2.135871839.1553426731; _gid=GA1.2.1768504745.1554392662;
count=5; news=1537660800000; version=1548019000;
_gauges_unique_hour=1; _gauges_unique_day=1; _gauges_unique=1"
```

Ejemplos:

```
alert(document.cookie);
document.cookie = "nombre = Ada";
```

Crear una nueva cookie

```
document.cookie = nuevaCookie;
```

En el código anterior, `nuevaCookie` es una cadena con la forma `clave = valor`.
Solo se puede establecer/actualizar una cookie de cada vez utilizando este método.

Después del valor de la cookie se pueden especificar algunos atributos de la cookie. Se escriben con un punto y coma y después el nombre del atributo y su valor:

- `path = path;` (ej. path='/'). Si no se especifica, se toma como valor por defecto el path del documento actual.
- `max-age = max-age-en-segundos;`
- `expires = fecha-en-formato-GMT;`
  Si no es especifican `max-age` o `expires`, la cookie expira al terminar la sesión.

Ejemplo:

```
document.cookie = "nombre = pepe; max-age = 3600";
```

Modificar una cookie
Modificar una cookie es sobrescribirla.

Ejemplo:

```
document.cookie = "nombre = Laura";
```

Borrar una cookie

Para borrar una cookie se puede dar una fecha de expiración anterior a la actual.

Ejemplo:

```
document.cookie = "nombre =; expires=Thu, 01 Jan 1970 00:00:00 UTC";
```

Obtener el valor de una cookie

Para obtener el valor de una cookie individual, hay que:

- Obtener una cadena con todas las cookies y sus valores (document.cookie).
- Localizar, en esa cadena, la cookie que nos interesa y su valor.

Ejemplo[1]: Asignar a `valorCookie` el valor de la cadena con nombre `miCookie`

```
let valorCookie = document.cookie
  .split('; ')
  .find(cadena => cadena.startsWith(miCookie='))
  .split('=')[1];
```

[1] https://devdocs.io/dom/document/cookie

Web Storage

JavaScript soporta persistencia de datos en el navegador a través de Web Storage. Esta técnica, nos permite almacenar más información y de forma más intuitiva y segura que con las cookies. Otra ventaja de Web Storage es que almacena los datos por origen, es decir, todas las páginas con el mismo origen (protocolo, dominio y puerto), podrán acceder a los mismos datos. Por estos motivos, hoy en día **se recomienda utilizar esta técnica en vez de cookies**.

Disponemos de dos mecanismos distintos para trabajar con Web Storage, ambos disponibles con las siguientes propiedades del objeto `Window`:

- **`localStorage`**: permite crear contenedores de datos permanentes que solo se eliminan si se borran desde JavaScript.
- **`sessionStorage`**: permite crear contenedores de datos asociados a la sesión.
  - Comienzo de sesión: apertura de navegador o pestaña.
  - Final de sesión: cierre de navegador o pestaña.

Ambas propiedades deben almacenar siempre strings, no pueden almacenar otro tipo de datos.

Los métodos que se pueden utilizar para gestionar los datos almacenados son:

Establecer un item: .setItem()

```
localStorage.setItem("nombre", "valor");
sessionStorage.setItem("nombre", "valor");
```

También se puede hacer mediante propiedades dinámicas, pero hoy en día no se recomienda:

```
localStorage.nombre = "valor";
sessionStorage.nombre = "valor";
```

Obtener el valor de un ítem: .getItem()

```
localStorage.getItem("nombre");
sessionStorage.getItem("nombre");
```

También se puede hacer mediante propiedades dinámicas, pero hoy en día no se recomienda:

```
localStorage.nombre;
sessionStorage.nombre;
```

Eliminar un ítem: .removeItem()

```
localStorage.removeItem("nombre");
sessionStorage.removeItem("nombre");
```

Eliminar todos los ítems: .clear()

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

Comprobar si el navegador soporta Storage

```
if (typeof(Storage) != "undefined") {
      //Soporta Web Storage
}
```

Same Origin Policy

Hemos de tener en cuenta que los contenedores de `localStorage` y `sessionStorage` siguen la política de seguridad **same-origin-policy**, es decir, un script solo puede acceder a contenedores creados por otros scripts que vinieron del mismo origen, es decir, del mismo servidor. El origen de un script son el protocolo, dominio y puerto del servidor.

La siguiente tabla[2] muestra ejemplos de comparaciones de origenes para la URL `http://store.company.com/dir/page.html`:

URL	Resultado	Razón
http://store.company.com/dir2/other.html	Mismo origen	Solo la ruta difiere
http://store.company.com/dir/inner/another.html	Mismo origen	Solo la ruta difiere
https://store.company.com/secure.html	Fallo	Diferente protocolo
http://store.company.com:81/dir/etc.html	Fallo	Diferente puerto
http://news.company.com/dir/other.html	Fallo	Diferente host

Si se viola esta política de seguridad, el navegador lanzará la excepción ***security error*** o simplemente no funcionará correctamente. Esto suele ocurrir siempre que trabajemos con archivos locales. En ocasiones lo podemos solucionar accediendo por medio de localhost en vez de a través de la ruta a nuestro archivo, pero la solución real es alojar nuestros archivos en servidores, por ejemplo: https://neocities.org/ nos ofrece servicio gratuito.

También hemos de tener en cuenta que el usuario puede tener su navegador configurado para denegar permisos a datos persistentes para el origen especificado, en ese caso no podemos hacer nada, salvo pedirle que los permita.

[2]Tomada de: https://developer.mozilla.org/es/docs/Web/Security/Same-origin_policy

UNIDAD 9 (4) - Formularios - DOMContentLoaded

Desarrollo Web en Entorno Cliente

2º DAW

Contenidos

Eventos DOMContentLoaded y load

Como hemos ido viendo crear nuestros scripts, nos interesa que el código se ejecute cuando se termine de cargar la página completamente, o al menos, el árbol DOM.

Para ello, podemos utilizar el evento `DOMContentLoaded`. En `miFuncion` estará todo el código que queremos que se ejecute cuando se termine de carga el DOM.

```
document.addEventListener("DOMContentLoaded", miFuncion);
```

El evento **`DOMContentLoaded`** indica cuando finaliza la carga de la página, aunque los demás recursos (imágenes, ficheros JavaScript o CSS, etc.) no hayan cargado todavía. El evento **`load`** ocurre en cambio cuando se ha cargado la página y todos sus recursos, por lo tanto, `DOMContentLoaded` ocurre antes que `load` y es preferible utilizarlo.

Eventos HTML

Evento	Descripción
`load`	Se produce en el objeto `window` cuando la página y todos sus recursos, se cargan por completo. En el elemento <`img`> cuando se carga por completo la imagen.
`DOMContentLoaded`	Se produce cuando finaliza la carga de la página, aunque los demás recursos (imágenes, ficheros JavaScript o CSS, etc.) no hayan cargado todavía, pero el árbol DOM sí.

UT 10 – SOLICITUDES DE RED (RESUMEN)

Contenido

1. JSON[1]

JSON es un estándar que permite representar valores y objetos en una cadena. Este estándar es reconocido por muchos lenguajes (JavaScript, PHP, Ruby, Java…).

Se utiliza para:

- Intercambiar datos entre distintos lenguajes si estos lenguajes reconocen el mismo formato JSON. Un caso en el que se utiliza es para intercambiar datos con el servidor.
- Almacenar datos en formato cadena. En algunos casos, sólo se pueden almacenar cadenas. Utilizando el formato JSON podríamos almacenar cualquier valor.

Qué valores se pueden representar en cadenas JSON:

- Un valor primitivo: número, boolean, cadena
- Un objeto
- Un array
- El valor `null`

Estos valores pueden estar anidados (por ejemplo, un array cuyos elementos son objetos).

Los siguientes valores **no** se pueden serializar:

- Una función, un método
- El valor undefined
- El tipo de datos SYMBOL

Ejemplo: el siguiente valor

```
let student = {  name: 'John',
                 age: 30,
                 isAdmin: false,
                 courses: ['html', 'css', 'js'],
                 wife: null
               };
```

Se puede representar con el siguiente formato JSON:

```
{  "name": "John",
   "age": 30,
   "isAdmin": false,
   "courses": ["html", "css", "js"],
   "wife": null
}
```

Vemos que:

- Las propiedades y las cadenas se escriben entre dobles comillas.
- Los valores numéricos, booleanos y null se escriben directamente, sin comillas.
- Los arrays se representan entre corchetes [ ] *(como en JavaScript)*.
- Los objetos se representan entre llaves { } *(como en JavaScript)*

[1] Más detallado en: https://es.javascript.info/json

Obtener la cadena JSON a partir de un valor (serializar)

```
cadena = JSON.stringyfy( valor )
```

Este método devuelve una cadena que representa al `valor` en formato JSON.

Ejemplo:

```
let student = {  name: 'John',
                 age: 30,
                 isAdmin: false,
                 courses: ['html', 'css', 'js'],
                 wife: null
              };

let cadenaJSON = JSON.stringify(student);

alert(typeof cadenaJSON); // ¡obtenemos un string!

alert(cadenJSON);
/*{  "name": "John",
   "age": 30,
   "isAdmin": false,
   "courses": ["html", "css", "js"],
   "wife": null
}
*/
```

Obtener un valor a partir de la cadena JSON

```
valor = JSON.parse( cadena )
```

Este método obtiene un dato a partir de una cadena que representa un `valor` en formato JSON.

Ejemplo: Seguimos con el ejemplo anterior.

```
estudiante2 = JSON.parse(cadenaJSON);
alert( estudiante2.name);              // John
```

Cuidado con las fechas

Ojo! Se puede serializar una fecha, pero para recuperar su valor, tendremos que crear un objeto Date a partir de la cadena que devuelve JSON.

Ejemplo: En este ejemplo, creamos un objeto de tipo fecha (`hoy`). Lo serializamos, y luego intentamos recuperar su valor. En `fecha1` recuperamos una cadena, no una fecha. Sin embargo, en `fecha2` vemos que se ha convertido la fecha en una cadena.

```
let hoy = new Date()
hoy_JSON = JSON.stringify(hoy) // "\"2022-02-01T08:40:35.826Z\""

let fecha1 = JSON.parse(hoy_JSON)             // String
let fecha2 = new Date(JSON.parse(hoy_JSON))   // Objeto Date
```

2. PROMESAS

Las promesas son objetos de JavaScript 6 (ES2015) que nos van a permitir gestionar operaciones asíncronas.

Las promesas nos van a servir para hacer llamadas a servidor y recibir datos del servidor sin necesidad de recargar la página. Van a comportarse como objetos intermediarios entre el código del cliente y el código del servidor.

En el siguiente resumen, sólo se verán algunas características de las promesas, para centrarnos en su utilización en las llamadas al servidor.

Crear una promesa[2]

Los objetos de tipo promesa se crean de este modo:

```
let promise = new Promise(function(resolve, reject) {
  // Ejecutor (el código productor, "cantante")
});
```

resolve y reject son dos funciones que pasa automáticamente JavaScript a la función. No hay que crearlas.

Dentro de la función está el código que se quiere ejecutar. Cuando el código obtiene un valor correcto, dentro de la función se llama a resolve(), pudiendo pasar un valor.

Si en la ejecución se quiere dar algún error, se llama a reject(), pudiendo pasar un valor como parámetro.

Ejemplo de promesa que termina correctamente: La promesa se ejecuta cuando se construye la promesa. Al cabo de 1 segundo, indica que la tarea está realizada, con el resultado "hecho".

```
let promise = new Promise(
     function(resolve, reject) {
          setTimeout(() => resolve("hecho"), 1000);
     }
);
```

Ejemplo de promesa que termina con error: Ejemplo similar al anterior, pero en este caso se quiere indicar que la tarea ha tenido un error, y se crea un objeto de tipo error con el texto "¡Vaya".

```
let promise = new Promise(
     function(resolve, reject) {
          setTimeout(() => reject(new Error("¡Vaya!")), 1000);
     }
);
```

El objeto promise tiene estas propiedades internas (no se puede acceder desde fuera de la promesa):

- state: puede tener los siguientes valores:
  - pending: pendiente
  - fulfilled: cumplido (cuando se llama a resolve)
  - rejected: rechazado (cuando se llama a reject)
- result: puede tener los siguientes valores:
  - undefined: valor inicial

[2] https://es.javascript.info/promise-basics

- *valor*: si se llama a resolve(*valor*)
- *error*: si se llama a rejected(*error*)

Tener en cuenta:

- La función que pasamos a la promesa se ejecuta automáticamente cuando se crea el objeto de tipo promesa.
- Sólo puede haber un único resultado o un error. Todas las llamadas adicionales a resolve() y reject() son ignoradas.
- resolve() y reject() sólo esperan un argumento (o ninguno) e ignorarán los argumentos adicionales.
- Cuando se llama a reject(), se recomienda usar un objeto de tipo Error.

Consumidores: then, catch, finally

Las promesas tienen unos métodos (.then(), .catch() y .finally().) donde se registran las funciones que se quieren ejecutar cuando la promesa pase al estado fulfilled (terminado correctamente), rejected (rechazado), o cualquiera de los dos.

.then(funciónOk, funciónError)

La sintaxis es:

```
promise.then(
  function(result) { /* manejar un resultado exitoso */ },
  function(error) { /* manejar un error */ }
);
```

El primer argumento es una función que se ejecuta cuando la promesa ha terminado con resolve(). result es el valor pasado a resolve().

El segundo argumento es una función que se ejecuta cuando la promesa ha terminado con reject().error es el valor pasado a resolve().

Ejemplo:

```
let promise = new Promise(function(resolve, reject) {
  setTimeout(() => reject(new Error("Vaya!")), 1000);
});

// reject ejecuta la segunda función en .then
promise.then(
  result => alert(result), // no se ejecuta
  error => alert(error) // muestra "Error: ¡Vaya!" después de 1 segundo
);
```

.catch(funciónError)

Permite registrar una función que se ejecutará si se produce un error

```
promise.catch(
  function(error) { /* manejar un error */ }
);
```

La llamada .catch(f) es un análogo completo de .then(null, f), es solo una abreviatura.

.finally(función)

Función que se ejecuta siempre, se resuelva la promesa o se rechace.

1. Un manejador `finally` no tiene argumentos. En `finally` no sabemos si la promesa es exitosa o no. Eso está bien, ya que nuestra tarea generalmente es realizar procedimientos de finalización "generales".
2. Un manejador `finally` traspasa resultados y errores al siguiente manejador.

Encadenamiento de promesas

El método `promise.then` devuelve una nueva promesa.

Si la función (manejador) que se pasa al then:

- devuelve un valor: la promesa devuelta por el then, se considera resuelta con el valor devuelto como su valor (como si se hubiera hecho resolve(valor).
- no devuelve nada: la promesa devuelta por el then, se considera resuelta con el valor undefined.
- se produce un error: la promesa devuelta por el then, se considera rechazada con el error como su valor.
- devuelve una promesa resuelta: la promesa devuelta por el then se considera finalizada con el valor de la promesa como su valor.
- devuelve una promesa rechazada: la promesa devuelta por el then se considera rechazada con el valor de la promesa como su valor.
- devuelve una promesa pendiente de resolver: el manejador espera a que se estabilice y luego obtiene su resultado; la resolución o el rechazo de la promesa devuelta por el then será lo que corresponda a la resolución/rechazo de la promesa.

Ejemplo1: Encadenando promesas que devuelven valores.

```
new Promise(function(resolve, reject) {

  setTimeout(() => resolve(1), 1000); // (*)

}).then(function(result) { // (**)

  alert(result); // 1
  return result * 2;

}).then(function(result) { // (***)

  alert(result); // 2
  return result * 2;

}).then(function(result) {

  alert(result); // 4
  return result * 2;

});
```

Ejemplo 2: Encadenando promesas que devuelven promesas.

```
new Promise(function(resolve, reject) {

  setTimeout(() => resolve(1), 1000);

}).then(function(result) {

  alert(result); // 1

  return new Promise((resolve, reject) => { // (*)
    setTimeout(() => resolve(result * 2), 1000);
  });

}).then(function(result) { // (**)

  alert(result); // 2

  return new Promise((resolve, reject) => {
    setTimeout(() => resolve(result * 2), 1000);
  });

}).then(function(result) {

  alert(result); // 4

});
```

async/await

JavaScript 8 (ES2017) ha proporcionado una sintaxis especial para trabajar con promesas de una forma más confortable, llamada `"asyn/await"`.

Funciones async

La palabra reservada `async` se coloca delante de la declaración de una función. De este modo se indica que la función siempre devolverá una promesa. Otros valores serán envueltos y resueltos en una promesa automáticamente.

Ejemplo 1: Si la función devuelve un valor, ese valor se considera el valor con el que se resuelve la promesa.

```
async function f() {
  return 1;
}
f().then(alert);    // 1
```

Ejemplo 2: La función puede devolver una promesa. El siguiente ejemplo es equivalente al anterior.

```
async function f() {
  return Promise.resolve(1);
}

f().then(alert); // 1
```

await

Esta palabra reservada sólo puede utilizarse dentro de funciones async.

`await` hace que JavaScript espere a que una promesa responda y devuelve su resultado.

Ejemplo:

```
async function f() {

  let promise = new Promise((resolve, reject) => {
    setTimeout(() => resolve("¡Hecho!"), 1000)
  });

  let result = await promise; // espera hasta que la promesa se resuelva (*)

  alert(result); // "¡Hecho!"
}

f();
```

La ejecución de la función es pausada en la línea (*) y se reanuda cuando la promesa se resuelve devolviendo un valor que se asigna a `result`. Ese valor se muestra con un `alert`.

Se podría haber hecho también con un `then`:

```
function f() {

  let promise = new Promise((resolve, reject) => {
    setTimeout(() => resolve("¡Hecho!"), 1000)
  });
```

```
  promise.then( function(result) {
       alert(result);
  }
}

f();
```

`async`/`await` nos proporcionan una sintaxis más sencilla para tener el resultado de una promesa que promise.`then`.

Manejo de errores con await

Si una promesa se resuelve, `await` `promise` devuelve su resultado. En caso de error se dispara un erro que puede ser capturado con un `try..catch`. En el bloque `try..catch` puede haber varios `await`.

Si no tenemos `try..catch`, entonces la promesa generada por `async f()` se vuelve rechazada.

Ejemplo: llamamos a `fetch` que devuelve una promesa con los datos para acceder al servidor, o devuelve un error si no se ha podido hacer la conexión con el servidor.

```
async function f() {

  try {
    let response = await fetch('http://no-such-url');
  } catch(err) {
    alert(err); // TypeError: failed to fetch
  }
}

f();
```

Ejemplo 2: en el bloque `try` podría haber varios `await`.

```
async function f() {

  try {
    let response = await fetch('/no-user-here');
    let user = await response.json();
  } catch(err) {
    // atrapa errores tanto en fetch como en response.json
    alert(err);
  }
}

f();
```

Ejemplo 3: Cómo gestionar los errores si no se utiliza el bloque `try..catch`.

```
async function f() {
  let response = await fetch('http://no-such-url');
}

// f() se vuelve una promesa rechazada
f().catch(alert); // TypeError: failed to fetch // (*)
```

3. PETICIONES AL SERVIDOR CON PROMESAS: FETCH[3]

La función fetch

fetch() es un método global que inicia el proceso de obtener un recurso de la red, devolviendo una promesa que es resuelta cuando la respuesta está disponible.

La promesa se resuelve con un objeto de tipo Response que representa la respuesta a la petición.

La promesa se resuelve aunque haya errores HTTP (ej. 404). Por tanto, el manejador del then() debe comprobar Response.ok y/o Response.status.

La promesa sólo termina con un rechazo si se encuentra un error al acceder al servidor (ej. el servidor no existe, problemas de red...)

Sintaxis

```
let promise = fetch(url, [options])
```

donde:

- **`url`** – representa la dirección URL a la que deseamos acceder.
- **`options`** – objeto que contiene la configuración que se quiere aplicar a la conexión. Si se omite, se ejecuta una petición GET.

  Algunas de las propiedades que puede tener ese objeto son:

  - method: "GET", "POST", "PUT"; "DELETE", ..
  - headers: cabeceras que se quieren enviar.
  - body: un cuerpo que se quiere añadir a la petición. Sólo para peticiones POST.
  - mode: puede ser "cors", "no-cors" o "same-origin"
  - credentials: controla que tiene que hacer el navegador con las credenciales. Puede ser una de las siguientes cadenas:
    - "omit"
    - "same-origin"
    - "include"

Cómo hacer una petición al servidor

Obtener una respuesta es un proceso de dos pasos:

1. Llamada a **fetch()**. Fech devuelve una promesa que se resuelve con un objeto de la clase Response tan pronto como el servidor responde con los encabezados de la petición.
   Este objeto tiene dos valores que nos permiten conocer el resultado de la petición hecha al servidor:
   - **status**: Se puede chequear para comprobar el código de estado HTTP (ej. 200)
   - **ok**: true si el código de estado es 200 a 299

2. Obtener el cuerpo de la respuesta. Si no se ha producido error http, (response.ok==true), se puede llamar a un método de la respuesta, como:

[3] Más detallado en: https://es.javascript.info/fetch

- **`response.text()`** – devuelve una promesa que se resuelve con la representación en texto del cuerpo de la respuesta.
- **`response.json()`** – devuelve una promesa que se resuelve con el resultado de parsear el texto del cuerpo de la respuesta como un JSON.

IMPORTANTE

- Podemos elegir un solo método de lectura para el cuerpo de la respuesta.
- Si ya obtuvimos la respuesta con `response.text()`, entonces `response.json()` no funcionará, dado que el contenido del cuerpo ya ha sido procesado.

Petición GET que devolverá un JSON

Ejemplo: hacer una llamada a una url que devolverá un json;

```
let response = await fetch(url);
if (response.ok) { // si el HTTP-status es 200-299
  let json = await response.json();
} else {
  alert("Error-HTTP: " + response.status);
}
```

Petición GET enviando parámetros

Hay que agregar los parámetros a la URL como `?nombre=valor`, y asegurar la codificación adecuada. Podemos hacerlo manualmente (creando una cadena) o utilizando el objeto URL:

```
let url = new URL('https://google.com/search');
url.searchParams.set('q', 'pruébame!');

// el parámetro 'q' está codificado
xhr.open('GET', url); // https://google.com/search?q=test+me%21
```

Petición POST enviando un JSON

En este apartado vemos un ejemplo de cómo enviar datos de tipo JSON.

Ejemplo: enviar información al servidor como un objeto JSON.

```
let user = {
  nombre: 'Juan',
  apellido: 'Perez'
};

let response = await fetch('/article/fetch/post/user', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json;charset=utf-8'
  },
  body: JSON.stringify(user)
});

let result = await response.json();
alert(result.message);
```

Recuperar los datos en el servidor

```
<?php
$json = file_get_contents('php://input');
$data = json_decode($json);

$nombre = $data->nombre;
$apellido = $data->apellido;

echo "El usuario " . $nombre." ".$apellido . " ha sido registrado." ;
?>
```

Petición POST enviando datos con FormData[4]

Los objetos FormData pueden facilitar la tarea de enviar datos al servidor.

Crear un FormData

```
let formData = new FormData([form]);
```

Si le pasamos un elemento HTML de tipo formulario, el objeto automáticamente capturará sus campos.

Envío de un formulario simple

Ejemplo 1: En el siguiente ejemplo vemos cómo se pueden enviar al servidor los datos de un formulario simple.

```
<form id="formElem">
  <input type="text" name="name" value="John">
  <input type="text" name="surname" value="Smith">
  <input type="submit">
</form>

<script>
  formElem.onsubmit = async (e) => {
    e.preventDefault();

    let response = await fetch('recibirUsuario.php', {
      method: 'POST',
      body: new FormData(formElem)
    });

    let result = await response.json();

    alert(result.message);
  };
</script>
```

Cómo se pueden recuperar los datos en un programa php:

[4] https://es.javascript.info/formdata

```php
<?php
$nombre = $_POST["name"];
$apellido = $_POST["surname"];

echo $nombre." ".$apellido;
?>
```

Ejemplo 2: Los campos <input type="file"> también son enviados, como sucede en un envío normal.

Métodos de FormData

Existen métodos para modificar los campos del FormData:

- `formData.append(name, value)` – agrega un campo al formulario con el nombre name y el valor `value`,
- `formData.append(name, blob, fileName)` – agrega un campo tal como si se tratara de un `<input type="file">`, el tercer argumento fileName establece el nombre del archivo (no el nombre del campo), tal como si se tratara del nombre del archivo en el sistema de archivos del usuario,
- `formData.delete(name)` – elimina el campo de nombre `name`,
- `formData.get(name)` – obtiene el valor del campo con el nombre `name`,
- `formData.has(name)` – en caso de que exista el campo con el nombre `name`, devuelve `true`, de lo contrario `false`

Un formulario técnicamente tiene permitido contar con muchos campos con el mismo atributo `name`, por lo que múltiples llamadas a append agregarán más campos con el mismo nombre.

Por otra parte existe un método `set`, con la misma sintaxis que `append`. La diferencia está en que `.set` remueve todos los campos con el name que se le ha pasado, y luego agrega el nuevo campo. De este modo nos aseguramos de que exista solamente un campo con determinado `name`, el resto es tal como en `append`:

- `formData.set(name, value)`,
- `formData.set(name, blob, fileName)`.

También es posible iterar por los campos del objeto formData utilizando un bucle `for..of`:

```
let formData = new FormData();
formData.append('key1', 'value1');
formData.append('key2', 'value2');

// Se listan los pares clave/valor
for(let [name, value] of formData) {
  alert(`${name} = ${value}`);                                    // key1 = value1,
luego key2 = value2
}
```

4. PETICIONES AL SERVIDOR CON XMLHttpRequest[5]

`XMLHttpRequest` es un objeto nativo del navegador que permite hacer solicitudes HTTP desde JavaScript.

A pesar de tener la palabra "XML" en su nombre, se puede operar sobre cualquier dato, no solo en formato `XML`. Podemos cargar/descargar archivos, dar seguimiento y mucho más.

Ahora hay un método más moderno `fetch` que en algún sentido hace obsoleto a `XMLHttpRequest`.

En el desarrollo web moderno `XMLHttpRequest` se usa por tres razones:

1. Razones históricas: necesitamos soportar scripts existentes con `XMLHttpRequest`.
2. Necesitamos soportar navegadores viejos, y no queremos polyfills (p.ej. para mantener los scripts pequeños).
3. Necesitamos hacer algo que `fetch` no puede todavía, ej. rastrear el progreso de subida.

¿Te suena familiar? Si es así, está bien, adelante con `XMLHttpRequest`. De otra forma, por favor, dirígete a `Fetch`.

Lo básico

Para hacer la petición, necesitamos seguir varios pasos:

1. Crear el objeto XMTHttpRequest

```
let xhr = new XMLHttpRequest();
```

2. Inicializarlo, normalmente justo después de crearlo.

```
xhr.open(method, URL, [async, user, password])
```

siendo:

- `method`: Método HTTP. Normalmente, "GET" o "POST"
- `URL`
- `async`: si la petición será síncrona o no (por defecto es asíncrona, valor utilizado casi siempre)
- `user`, `password`: usuario y contraseña para atenticación HTTP básica

La llamada a `open`, no abre la conexión. Sólo configura la solicitud, pero la actividad de red sólo comienza con la llamada al método `send`.

3. Establecer el formato de respuesta esperada *(SÓLO EN ALGUNOS CASOS, por ejemplo, si la respuesta no es texto).*

```
xhr.responseType = "..."
```

Posibles tipos:

- `""` (default) – obtiene una cadena,
- `"text"` – obtiene una cadena,
- `"arraybuffer"` – obtiene un ArrayBuffer
- `"blob"` – obtiene un Blob

[5] https://es.javascript.info/xmlhttprequest

- `"document"` – obtiene un documento XML (puede usar XPath y otros métodos XML) o un documento HTML (en base al tipo MIME del dato recibido),
- `"json"` – obtiene un JSON (automáticamente analizado).

4. Enviar cabeceras *(SÓLO EN ALGUNOS CASOS, por ejemplo, si enviamos datos json).*

```
xhr.setRequestHeader(nombre, valor);
```

5. Enviar la petición.

```
xhr.send([body])
```

Este método abre la conexión y envía la solicitud al servidor. El parámetro adicional body contiene el cuerpo de la solicitud.

Algunos métodos como GET no tienen un cuerpo. Y otros como POST usan el parámetro body para enviar datos al servidor.

6. Escuchar los eventos de respuesta del objeto

Los eventos más comunes son:

- `xhr.onload = function()...` – cuando la solicitud está; completa (incluso si el estado HTTP es 400 o 500), y la respuesta se descargó por completo.
  Una vez el servidor haya respondido, podemos recibir el resultado en las siguientes propiedades de xhr:
  - **`xhr.status:`** Código del estado HTTP (un número): 200, 404, 403, etc., puede ser 0 en caso de un fallo no HTTP.
  - **`xhr.statusText:`** Mensaje del estado HTTP (una cadena): usualmente OK para 200, Not found para 404, Forbidden para 403.
  - **`xhr.response`** (scripts antiguos utilizan `responseText`)
- `xhr.onerror = function()...` – cuando la solicitud no pudo ser realizada satisfactoriamente, ej. red caída o una URL inválida.
- `xhr.onprogress = function()...` – se dispara periódicamente mientras la respuesta está siendo descargada, reporta cuánto se ha descargado.

Ejemplo1: obtener una respuesta de texto

```
let xhr = new XMLHttpRequest();

xhr.open('GET', '/article/xmlhttprequest/example/json');

xhr.send();

xhr.onload = function() {
   if (xhr.status != 200) {
       alert(`Error ${xhr.status}: ${xhr.statusText}`);
```

```
   } else {
       texto = xhr.response;
       alert(texto);
   }
}

xhr.onerror = function() { // solo se activa si la solicitud no se puede realizar
  alert(`Error de red`);
};
```

Métodos y propiedades obsoletas

En los scripts antiguos se pueden encontrar las propiedades xhr.responseText y xhr.responseXML. Existen por razones históricas, pero hoy en día, debemos seleccionar el formato en xhr.responseType y obtener la respuesta en xhr.response.

xhr.onreadystatechange se utilizaba cuando no existían onload y otros eventos. Hoy load/error/progress lo hacen obsoleto.

Abortar solicitudes

Se puede terminar la solicitud en cualquier momento.

```
xhr.abort();  // termina la solicitud
```

Peticiones POST

Peticiones POST con formularios: FormData

Se puede crear un objeto FormData desde un formulario o manualmente. Después se envía al servidor. No es necesario enviar cabeceras porque el tipo de datos enviado es `multipart/form-data`.

```
<form name="person">
  <input name="name" value="John">
  <input name="surname" value="Smith">
</form>

<script>
  // pre llenado del objeto FormData desde el formulario
  let formData = new FormData(document.forms.person);

  // agrega un campo más
  formData.append("middle", "Lee");

  // lo enviamos
  let xhr = new XMLHttpRequest();
  xhr.open("POST", "/article/xmlhttprequest/post/user");
  xhr.send(formData);

  xhr.onload = () => alert(xhr.response);
</script>
```

Peticiones POST con JSON

En este caso, sí que hay que enviar cabeceras.

```
let xhr = new XMLHttpRequest();

let json = JSON.stringify({
  name: "John",
  surname: "Smith"
});

xhr.open("POST", '/submit')
xhr.setRequestHeader('Content-type', 'application/json; charset=utf-8');

xhr.send(json);
```