



**Centro Integrado de Formación Profesional**

**AVILÉS**

Principado de Asturias

## **UNIDAD 6: COMPONENTES Y OBJETOS PREDEFINIDOS**

**DESARROLLO WEB EN ENTORNO CLIENTE**

**2º CURSO**

**C.F.G.S. DISEÑO DE APLICACIONES WEB**

## REGISTRO DE CAMBIOS

Versión	Fecha	Estado	Resumen de cambios
1.0	03/02/2025	Aprobado	Primera versión
1.1	17/03/2025	Aprobado	Corrección de errores

## ÍNDICE

ÍNDICE .....	1
UNIDAD 6: COMPONENTES Y OBJETOS PREDEFINIDOS.....	2
6.1 Objetos nativos del lenguaje .....	2
6.1.1 El objeto Date .....	2
6.1.2 El objeto Math .....	5
6.1.3 Cláusula with .....	7
6.2. Interacción con el navegador. Objetos predefinidos asociados.....	8
6.2.1 Objeto window.....	9
6.2.2 Objeto location .....	10
6.2.3 Objeto history .....	10
6.2.4 Objeto navigator .....	10
6.2.5 Temporizadores y cronómetros.....	11
6.3 Generación de texto y elementos HTML desde código.....	16
6.3.1 Generación de elementos HTML desde código JavaScript .....	16
6.3.2 Listas en React.....	16
6.4 Creación de nuevas ventanas. Comunicación entre ventanas.....	19
6.4.1 Gestión de ventanas en JavaScript .....	19
6.4.2 React router .....	21
6.4.3 Entendiendo el renderizado React .....	24
6.5 Gestión de la apariencia de la ventana.....	28
6.5.1 Apariencia de las ventanas.....	28
6.5.2 Estilos en React.....	28
6.5.3 Frameworks de Estilos en React: Material-UI .....	31
6.6 Interacción con el usuario .....	33
Gestión de estado con reducers.....	33
6.7 Mecanismos del navegador para el almacenamiento y recuperación de información.....	35
6.7.1 Cookies.....	35
6.7.2 Otros mecanismos de almacenamiento local.....	36
6.7.3 ReactContext .....	37
6.8 Depuración y documentación del código.....	40
ÍNDICE DE FIGURAS.....	42
BIBLIOGRAFÍA – WEBGRAFÍA .....	42

## UNIDAD 6: COMPONENTES Y OBJETOS PREDEFINIDOS

### 6.1 OBJETOS NATIVOS DEL LENGUAJE

#### 6.1.1 EL OBJETO DATE

Date representa una instantánea de un milisegundo concreto en el tiempo. En realidad, es el formato que permite utilizar una fecha con su hora correspondiente, llegando hasta el nivel del milisegundo. Por ejemplo, una representación completa de fecha podría ser la siguiente: AAAA-MM-DD HH:MM:SS.mmm (A - Año, M - Mes, D - Día, H - Hora, M - Minuto, S - Segundo, m - milisegundo).

Si se inicializa un objeto Date sin parámetros, JavaScript considera el instante actual en formato local.

```
let hoy = new Date(); // devuelve fecha y hora locales
```

Si se manda este comando a una consola, JavaScript muestra como resultado una versión del instante como cadena similar a la siguiente:

Tue Jun 25 2024 19:17:51 GMT+0200 (hora de verano de Europa central)

La creación de fechas puede realizarse de muy diversas formas. A continuación, puede verse un ejemplo de inicialización de variable de este tipo:

```
// Crea una nueva fecha desde cadena  
let nuevaFecha = new Date("2024-6-25 19:19:43");
```

Para mostrar una fecha con el formato local, se usa el método **toLocaleDateString()** de la clase **Date**.

Una vez que se tiene definida una variable que contiene un objeto de tipo Date sería interesante poder acceder a las diferentes partes de la fecha y hora de manera directa, es decir, saber directamente el día de la semana en el que se está, o sólo la hora, etc.

Para ello la clase Date cuenta con una serie de métodos que se enumeran a continuación:

- **getTime:** obtiene solamente la parte correspondiente a la hora, rechazando la fecha. Devuelve el número de segundos transcurridos desde el 1/1/ 1970. Es útil para crear marcas de tiempo para ciertas aplicaciones especiales de comunicación con el servidor.
- **getDay:** devuelve el día de la semana de la fecha, empezando a contar en domingo, que sería el día 0. De este modo el lunes sería el 1, el martes el 2 y así sucesivamente.
- **getDate:** obtiene el día del mes dentro de la fecha dada.
- **getMonth:** devuelve el número correspondiente al mes contenido en el objeto Date. Se comienza a contar en 0, que correspondería al mes de enero.
- **getFullYear:** devuelve el año contenido en la variable de tipo fecha. Esta función tiene un comportamiento algo extraño puesto que devuelve 2 cifras para años anteriores al 2000,

y 3 cifras para años posteriores al 2000. Es decir, empieza a contar los años en el 1900. Así, para el año 1980 devolverá 80, para 2016 devolverá 116 y para 1897 será el año -3 (negativo). Su uso no está recomendado y se prefiere el método siguiente que es consistente en lo que devuelve.

- **getFullYear**: devuelve el año contenido en la variable de tipo fecha. Se debe usar generalmente en sustitución de **getYear**, que se considera obsoleta.
- **getHours**: permite obtener el valor de la hora en una variable Date, sin incluir la fecha actual, ni los minutos ni los segundos.
- **getMinutes**: con este método se consiguen averiguar los minutos de la hora indicada.
- **getSeconds**: permite obtener la parte correspondiente a los segundos en una variable de hora.
- **getMilliseconds**: facilita el número de milisegundos después del segundo actual que está especificado dentro de una variable de tipo Date. Es decir, aunque la hora actual se muestre redondeada a 19:22:37, internamente hay unos cuantos milisegundos de precisión más que se pueden obtener, por lo que sería por ejemplo 19:22:37:849 (siendo 849 los milisegundos a mayores que devolverá esta función).

De acuerdo con las especificaciones ECMA-Script que normalizan el lenguaje, JavaScript dispone de funciones para el manejo de unidades temporales estándar, lo que se conoce como Universal Coordinated Time, UTC u Hora Universal Coordinada. Este baremo para medir la hora es más conocido por todo el mundo como GMT (o Greenwich Meridian Time, hora en el meridiano de Greenwich).

Prácticamente todas las funciones vistas en el apartado anterior para obtener las partes de un objeto Date se han implementado también para horas universales. Sólo hay que añadir la palabra "UTC" después de "get" en sus nombres. Así se tienen métodos como **getUTCDay**, **getUTCDate**, **getUTCMonth**, etc.

Estos métodos tienen en cuenta la diferencia horaria del sistema en el que se está respecto al meridiano de Greenwich (o a UTC 0), de forma que es posible que un método "get" de hora local y un método "get" de hora UTC ofrezcan resultados completamente diferentes. Por ejemplo, la fecha:

```
n = new Date(2024,0,1,0,15);
```

correspondiente a los 15 primeros minutos del día de Año Nuevo del año 2024 en España (UTC+1, si se ejecuta el código en un ordenador en este país), es realmente una hora menos en UTC 0. Por ello, esa fecha que en hora local sería ya el 2024, en hora UTC es todavía del año anterior, 2023.

Las funciones **toUTCString** y **toGMTString** son equivalentes y devuelven las fechas expresadas respecto a UTC 0 o al meridiano de Greenwich.

Así, el siguiente código:

```
var fecha = new Date(2024,0,1,0,15);  
alert(fecha + "\n" + fecha.toUTCString());
```

mostrará lo siguiente:

Mon Jan 01 2024 00:15:00 GMT+0100 (hora estándar de Europa  
central)  
Sun, 31 Dec 2023 23:15:00 GMT

#### 1. Fecha vs su correspondiente UTC

El método **getTimezoneOffset** se ocupa de informar de los minutos de diferencia existentes entre la hora local y la UTC/GMT, y puede resultar muy útil cuando la diferencia horaria entre donde está el servidor y los usuarios es importante.

Usando esta función se podrían escribir todas las funciones UTC que se acaban de ver con suma facilidad. Para el caso de España esta función devuelve el número -60 o -120, ya que hay una hora de diferencia en invierno, pero dos en verano con el horario de ahorro energético.

En realidad, esta función puede devolver casi cualquier valor pues existen países con diferencias horarias con UTC que no son horas completas. Por ejemplo, Venezuela está en UTC-04:30 (o sea, cuatro horas y media por detrás), y Nepal está en UTC+05:45 (5 horas y tres cuartos por delante).

JavaScript abstrae de todas estas dificultades y siempre devuelve la diferencia horaria correcta con código similar a este:

```
var fecha = new Date();  
alert(fecha.getTimezoneOffset());
```

Al igual que se pueden guardar en una variable fechas y sus partes a través de los métodos "get..." y "getUTC...", es posible igualmente fijar a posteriori estos valores. La clase Date posee unos métodos "set..." que están pensados para modificar el contenido de las variables de este tipo para reflejar fechas y horas diferentes.

Ninguno de los métodos que se verán a continuación afecta a la hora o fecha reales del sistema, sólo a los valores almacenados en las variables. De esta forma, es posible almacenar fechas arbitrarias en una variable de tipo temporal previamente definida.

A continuación, se detallan dichos métodos:

- **setDate:** permite asignar un nuevo día del mes a la fecha contenida en una variable.
- **setMonth:** asigna un nuevo valor para el mes en la fecha contenida en la variable. Recordar que los meses empiezan a contar desde 0 para enero.

- **setFullYear:** permite cambiar el valor del año en una fecha especificando los cuatro dígitos del año. Este método permite especificar opcionalmente el mes y el día del mes como segundo y tercer parámetro respectivamente.
- **setTime:** permite asignar una nueva hora pasando como argumento el número de milisegundos transcurridos desde la medianoche del 1/1/1970. No se utiliza demasiado por razones obvias.
- **setMilliseconds:** fija el número de milisegundos que pasan del segundo en la hora actual de una variable.
- **setSeconds:** permite fijar el número de segundos actuales en la variable. Opcionalmente se pueden especificar también los milisegundos como segundo parámetro.
- **setMinutes:** fija los minutos actuales en la hora contenida en una variable. Tiene como parámetros opcionales el número de segundos y el de milisegundos transcurridos.
- **setHours:** permite fijar la hora actual para una variable de tipo Date. Tiene como tres parámetros opcionales los minutos, segundos y milisegundos.

Del mismo modo que antes se vieron funciones "get..." para horas locales y "getUTC..." para horas GMT, en el caso de los métodos "set.." ocurre igual y existen las equivalentes referidas a UTC (setUTCDate, setUTCMonth, setUTCFullYear, etc.) que funcionan de la misma manera, pero refiriéndose sus parámetros a hora universal UTC y no a la hora local del sistema donde se ejecuten.

Gracias a estas funciones se pueden modificar fechas preexistentes y operar con éstas.

La resta de dos fechas se devuelve en milisegundos. El resto de operaciones se basan en que cada fecha internamente tiene milisegundos.

Para obtener:

- Días: Se divide entre el número de ms que tiene un día, es decir,  $24*60*60*1000$
- Meses: Esta vez se divide por  $30*24*60*60*1000$
- Años: Esta vez se divide por  $365*24*60*60*1000$

Para saber el número de milisegundos de una fecha, se puede usar **parse()** en Date

Formas de sumar fechas:

- Usar los milisegundos que se requieran e incrementarlos a la fecha (con **parse**)
- Usar métodos set. Por ejemplo, para sumar cinco días a una fecha base: **base.setDate(base.getDate()+5)**
- Para sumar segundos, se usan **setSeconds** y **getSeconds**

## 6.1.2 EL OBJETO MATH

Otro objeto interesante es Math, el cual es un objeto global que ofrece constantes y métodos para desarrollar funciones básicas para matemáticas.

A continuación, pueden verse algunos métodos de este objeto:

Método	Descripción	Ejemplo	
<b>abs</b>	Valor absoluto	Math.abs(-2)	2
<b>sin</b> <b>cos</b> <b>tan</b>	Funciones trigonométricas, reciben el argumento en radianes	Math.cos(Math.PI)	-1
<b>asin</b> <b>acos</b> <b>atan</b>	Funciones trigonométricas inversas	Math.asin(1)	1.57
<b>exp</b> <b>log</b>	Exponenciación y logaritmo, base E	Math.log(Math.E)	1
<b>ceil</b>	Devuelve el entero más pequeño mayor o igual al argumento	Math.ceil(-2.7)	-2
<b>floor</b>	Devuelve el entero más grande menor o igual al argumento	Math.floor(-2.7)	-3
<b>round</b>	Devuelve el entero más cercano o igual al argumento	Math.round(-2.7)	-3
<b>min</b> <b>max</b>	Devuelve el menor (o mayor) de sus dos argumentos	Math.min(2,4)	2
<b>pow</b>	Exponenciación, siendo el primer argumento la base y el segundo el exponente	Math.pow(2,3)	8
<b>sqrt</b>	Raíz cuadrada	Math.sqrt(25)	5
<b>random</b>	Genera un valor aleatorio comprendido entre 0 y 1.	Math.random()	0.7345

Las constantes que incluye la clase Math son:

Constante	Descripción
E	Constante matemática que representa al número <b>e</b> . Su valor es <b>2.71828...</b>
LN2	Constante matemática que representa al número resultado de calcular el <b>logaritmo neperiano de 2</b> . Su valor es <b>0.693...</b>
LN10	Constante matemática que representa al número resultante de calcular el <b>logaritmo neperiano de 10</b> . Su valor es <b>2.303...</b>
LOG10E	Valor del <b>logaritmo en base 10</b> de E.
PI	Constante matemática que representa al número <b>PI</b> . Su valor es <b>3.1416...</b>
SQRT1_2	Constante matemática que representa al número resultante de calcular la <b>raíz cuadrada de 1/2</b> . Su valor es <b>0.707...</b>
SQRT2	Constante matemática que representa al número resultante de calcular la <b>raíz cuadrada de 2</b> . Su valor es <b>1.414213</b>

Si se quisiera capturar el valor devuelto por algunos de estos métodos no habría más que asignárselo a una variable:

```
<script>  
    resultado = Math.sqrt(4)  
</script>
```

Para generar un valor aleatorio comprendido entre 1 y 10 hay que plantear lo siguiente:

```
let num = parseInt (Math.random() * 10) + 1;
```

Al multiplicar Math.random() por 10, genera un valor aleatorio comprendido entre un valor mayor a 0 y menor a 10, luego, con la función parseInt, se obtiene sólo la parte entera. Finalmente se suma 1.

### 6.1.3 Cláusula with

Dentro de with, basándose en un objeto (i.e. Math), en el interior de sus corchetes se puede acceder a las propiedades o métodos de ese objeto.

```
with (Math) {  
    res = (exp(x)-exp(y))*sin(y)/PI*log(z);  
}
```

Todas ellas son funciones de Math, pero no hace falta anteponer el objeto para llamarlas. Incluso en su interior se puede meter un return para una función. Lo que no se puede hacer es anidar elementos with.



## 6.2. INTERACCIÓN CON EL NAVEGADOR. OBJETOS PREDEFINIDOS ASOCIADOS

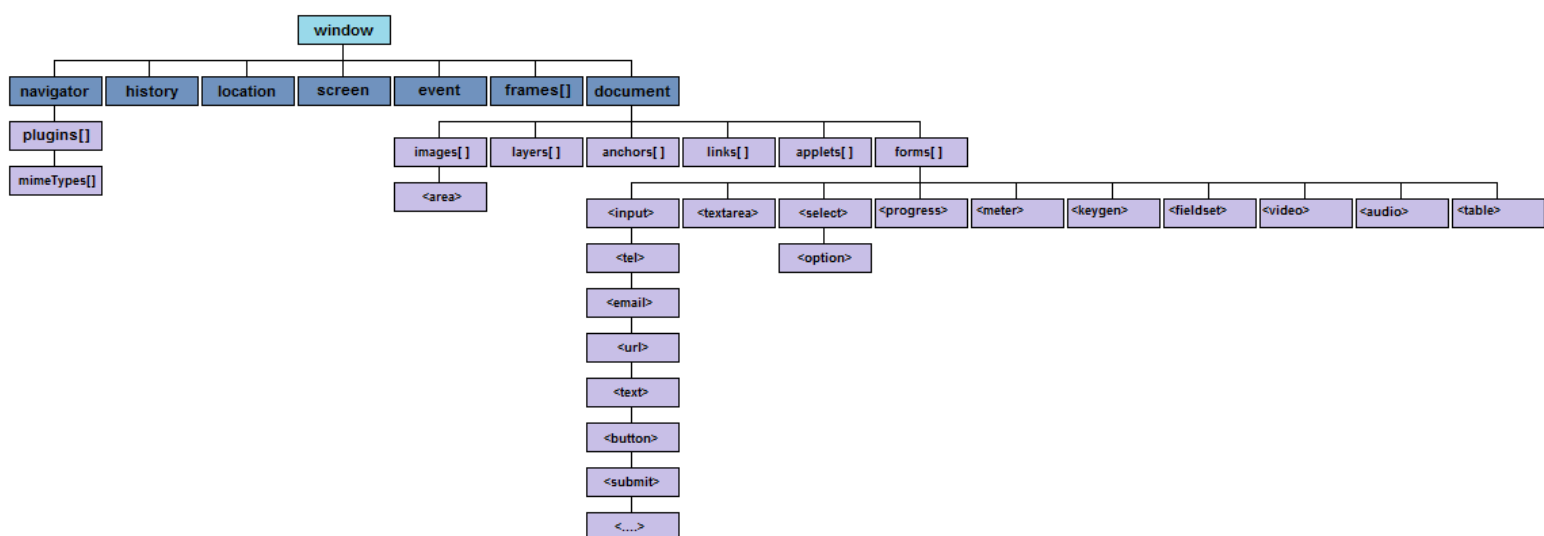
Anteriormente se vieron algunos de los principales objetos predefinidos de JavaScript. Sin embargo, existen otro tipo de objetos que permiten el control del navegador en sí mismo. Estos objetos permiten controlar diferentes aspectos del navegador como, el mensaje a mostrar en la barra de estado o como crear nuevas ventanas. Estos objetos están encuadrados en lo que se conoce como Browser Object Model o BOM. Al contrario que para el DOM, no existe un estándar de BOM, pero es bastante parecido en los diferentes navegadores.

Usando los objetos BOM es posible:

- Abrir, cambiar y cerrar ventanas
- Ejecutar código en cierto tiempo (temporizadores)
- Obtener información del navegador
- Ver y modificar propiedades de la pantalla
- Gestionar cookies, ...

Estos objetos son, entre otros:

- window
- navigator
- history
- location
- screen
- event
- iframe
- document



El soporte de navegador para todos estos objetos debe ser detectado antes de intentar acceder a sus propiedades.

Para detectar la compatibilidad de exploradores de un objeto, el enfoque genérico se deberá tener:

```
<script>
  if (document.object){
    // objeto soportado
  }
  else{
    // objeto no soportado
  }
</script>
```

La mayoría de los objetos están unidos mediante una jerarquía de objetos, donde el objeto superior es el actual window. Por ejemplo, el objeto forms se puede acceder como document.forms []. El objeto document también se accede como window.document, por lo que en el ejemplo anterior, la jerarquía completa es en realidad window.document.forms[], sin embargo, es perfectamente aceptable para omitir la referencia al objeto window.

### 6.2.1 OBJETO WINDOW

Representa la ventana del navegador y es el objeto principal. De hecho puede omitirse al llamar a sus propiedades y métodos, por ejemplo, el método **setTimeout()** es en realidad **window.setTimeout()**.

Sus principales propiedades y métodos son:

- .name: nombre de la ventana actual
- .screenX/.screenY: distancia de la ventana a la esquina izquierda/superior de la pantalla
- .outerWidth/.outerHeight: ancho/alto total de la ventana, incluyendo la toolbar y la scrollbar
- .innerWidth/.innerHeight: ancho/alto útil del documento, sin la toolbar y la scrollbar
- .open(url, nombre, opciones): abre una nueva ventana. Devuelve el nuevo objeto ventana.
- .opener: referencia a la ventana desde la que se abrió esta ventana (para ventanas abiertas con open)
- .close(): la cierra (pide confirmación, a menos que la hayamos abierto con open)
- .moveTo(x,y): la mueve a las coord indicadas
- .moveBy(x,y): la desplaza los px indicados
- .resizeTo(x,y): la da el ancho y alto indicados
- .resizeBy(x,y): le añade ese ancho/alto
- .scrollX / scrollY: scroll actual de la ventana horizontal / vertical

Otros métodos: .stop(), .focus(), .print(), ... NOTA: por seguridad no se puede mover una ventana fuera de la pantalla ni darle un tamaño menor de 100x100 px.

## Diálogos

Hay 3 métodos del objeto window que ya se han visto y que permiten abrir ventanas de diálogo con el usuario:

- `window.alert(mensaje)`: muestra un diálogo con el mensaje indicado y un botón de 'Aceptar'
- `window.confirm(mensaje)`: muestra un diálogo con el mensaje indicado y botones de 'Aceptar' y 'Cancelar'. Devuelve true si se ha pulsado el botón de aceptar del diálogo y false si no.
- `window.prompt(mensaje [, valor predeterminado])`: muestra un diálogo con el mensaje indicado, un cuadro de texto (vacío o con el valor predeterminado indicado) y botones de 'Aceptar' y 'Cancelar'. Si se pulsa 'Aceptar' devolverá un string con el valor que haya en el cuadro de texto y si se pulsa 'Cancelar' o se cierra devolverá null.

### 6.2.2 OBJETO LOCATION

Contiene información sobre la URL actual del navegador la cual puede ser modificada. Sus principales propiedades y métodos son:

- `.href`: devuelve la URL actual completa
- `.protocol`, `.hostname`, `.port`: devuelve el protocolo, host y puerto respectivamente de la URL actual
- `.pathname`, `hash`, `search`: devuelve la ruta al recurso actual, el fragmento (#...) y la cadena de búsqueda (?...) respectivamente
- `.reload()`: recarga la página actual
- `.assign(url)`: carga la página pasada como parámetro
- `.replace(url)`: ídem pero sin guardar la actual en el historial

### 6.2.3 OBJETO HISTORY

Permite acceder al historial de páginas visitadas y navegar por él:

- `.length`: muestra el número de páginas almacenadas en el historial
- `.back()`: vuelve a la página anterior
- `.forward()`: va a la siguiente página
- `.go(num)`: se mueve *num* páginas hacia adelante o hacia atrás (si *num* es negativo) en el historial

### 6.2.4 OBJETO NAVIGATOR

Proporciona información sobre el navegador y el sistema en que se ejecuta. Algunos de ellos son:

- `.userAgent`: muestra información sobre el navegador que usamos
- `.language`: muestra el idioma del navegador
- `.languages`: muestra los idiomas instalados en el navegador

También incluye objetos con sus propias API para poder interactuar con el sistema. A continuación, se enumeran algunos de ellos:

- `.geolocation`: devuelve un objeto con la localización del dispositivo (sólo funciona en https)
- `.storage`: permite acceder a los datos almacenados en el navegador
- `.clipboard`: permite copiar texto al portapapeles del usuario con `.writeText()` (sólo funciona en https)
- `.mediaDevices`: permite acceder a los dispositivos multimedia del usuario
- `.serviceWorker`: permite trabajar con *service workers*

Otros objetos que incluye BOM son:

- [`screen`](#): proporciona información sobre la pantalla
  - `.width/.height`: ancho/alto total de la pantalla (resolución)
  - `.availWidth/.availHeight`: igual pero excluyendo la barra del S.O.

### 6.2.5 TEMPORIZADORES Y CRONÓMETROS

Además de medir el tiempo, algo verdaderamente útil en cualquier plataforma de desarrollo es tener la capacidad de ejecutar código a intervalos regulares de tiempo. Un programa en JavaScript es una sucesión continua de líneas de código que se ejecutan una tras otra por orden para obtener un resultado. Sin embargo, en la realidad no solo llega con ejecutar el código en el momento de cargar una página, sino que es necesario que éste se ejecute en el momento apropiado: ante una acción concreta que ocurra en una página o al cabo de determinado tiempo. El primer caso se soluciona gracias a los eventos de HTML, los cuales permiten reaccionar ante diferentes situaciones que se den en una página (como pulsar un botón o pasar el cursor sobre un texto). La segunda situación se resuelve con el uso de temporizadores, y es el objeto de este apartado.

Los temporizadores son objetos sin representación física que se encargan de ejecutar una determinada tarea al cabo de un cierto tiempo. La orden para poner en marcha un temporizador es:

```
tempID = setTimeout(Rutina, Tiempo);
```

donde **Rutina** es el nombre de un procedimiento o función que el temporizador se encargará de ejecutar; y **Tiempo** es un valor expresado en milésimas de segundo que indica cuánto tiempo tardará éste en ejecutar la rutina especificada.

El nombre de una función a ejecutar es recomendable que vaya entre comillas por compatibilidad con navegadores antiguos, pero no es necesario.

En lugar del nombre de una función se le podría pasar también una cadena de texto con el código a ejecutar:

```
setTimeout("alert('Hola')", 2000);
```

Sin embargo, esto es considerado una mala práctica y solamente se soporta por compatibilidad con versiones antiguas del lenguaje. En la actualidad raramente se ve ya que, entre otras cosas, puede interferir con los procesos de minimización de código que se llevan a cabo para mejorar el rendimiento de descarga de las páginas.

La tercera opción es pasarle una función anónima que ejecute un determinado código:

```
setTimeout(function() { alert('Hola'); }, 2000)
```

Pero tampoco se utiliza a menudo, prefiriéndose por claridad la primera de las variedades de uso.

Al fijar un temporizador con **setTimeout** se puede anotar el valor devuelto en una variable. Este valor es un número que identifica de manera única al temporizador recién definido. El interés de este identificador reside únicamente en usarlo con el método complementario **clearTimeout** que lo toma como argumento para anular el temporizador relacionado en caso de que sea necesario cancelarlo antes de que se ejecute.

Según esto, si por ejemplo se dispone en el script de una función llamada **miFuncion** y se quiere activar ésta al cabo de 2 segundos habría que escribir:

```
let miId = setTimeout("miFuncion()", 2000);
```

En caso de que, por cualquier motivo, sea necesario detener este temporizador antes de que llegue a actuar y ejecute la función, habrá que ejecutar:

```
clearTimeout(miId);
```

Si se necesita ejecutar una función repetidamente, a intervalos regulares de tiempo, es posible llamar repetidas veces al temporizador de nuevo con el método que se acaba de ver o, mucho mejor, se puede usar el método **setInterval**. Éste es exactamente igual al anterior en cuanto a parámetros y forma de uso, pero la llamada se repite continuamente cada intervalo de tiempo especificado, hasta que se detenga con una llamada a su función complementaria **clearInterval**. Por ejemplo:

```
id = setInterval("Mifuncion()", 2000);
```

O bien:

```
id = setInterval(Mifuncion, 2000);
```

Lamará continuamente a la función indicada cada dos segundos. Sólo dejará de invocarla cuando se llame a **clearInterval** pasándole como parámetro el identificador devuelto al crear el temporizador (guardado en la variable "id" en la línea anterior):

```
clearInterval(id);
```

Tanto a **setTimeout** como a **setInterval** se les pueden pasar como parámetros opcionales los argumentos que necesite la función que se va a ejecutar. Tantos como sea necesario:

```
setTimeout(Mifuncion, 2000, parametro1, parametro2...);
```

Si se necesita pasar parámetros a la llamada a la función es mejor envolverla en otra función auxiliar sin parámetros y llamar a ésta.

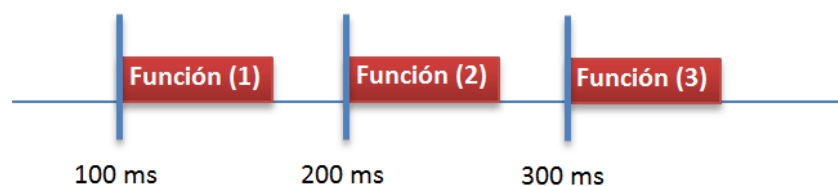
Hay que tener en cuenta que estos cuatro métodos de temporizadores no forman parte en realidad del lenguaje JavaScript, sino que son métodos proporcionados por los navegadores web (a través de su objeto global intrínseco **window**) y que pueden ser utilizados desde el código JavaScript incrustado en una página web. Si se quiere usar el código descrito fuera de un navegador no funcionará. Sin embargo, otros lenguajes basados en JavaScript suelen incluir su propia implementación nativa para facilitar la compatibilidad. Por ejemplo Node.js ofrece las cuatro funciones, que funcionan del modo descrito aquí.

El tiempo del intervalo para un temporizador puede ser cualquier valor. Sin embargo, si se pone demasiado pequeño, pueden ocurrir efectos no deseados. Aparte de poder llegar a bloquear la ejecución del navegador, a partir de determinado valor que varía de un navegador a otro (entre los 15 y los 25 milisegundos), el temporizador no tiene resolución suficiente y entonces da igual lo que se baje el tiempo puesto que no habrá diferencia. En cualquier caso, hay que evitar los procesos periódicos lanzados con intervalos muy pequeños. Todo lo que sea inferior a 100 milisegundos no parece muy recomendable salvo en casos muy puntuales, así que habría que mantenerse por encima de esa zona de confort.

Anteriormente se vio que **setInterval** llamaba a una función cada “x” milisegundos, especificados como segundo parámetro. Sin embargo, esto no es exactamente así, y en ocasiones puede tener importancia esta distinción.

JavaScript solamente posee un hilo de ejecución, por lo que todas las funciones se ejecutan dentro de éste (dejando de lado la salvedad de la API de Web Workers de HTML). Esto significa que no se pueden ejecutar dos scripts al mismo tiempo en una página. Los métodos **setTimeout** y **setInterval** ayudan a emular la ejecución en paralelo de código, ya que se pueden lanzar procesos al cabo de cierto tiempo o periódicamente, que parecen ejecutarse en otro hilo. Pero es solo una ilusión.

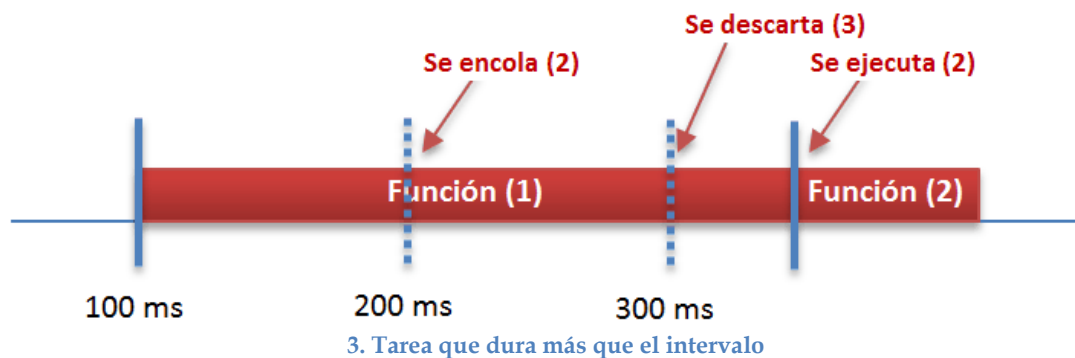
En realidad, tanto los intervalos de estos temporizadores como cualquier otro evento asíncrono que se provoque (a través de la interfaz de usuario, por ejemplo), se encolan para su ejecución para cuando el motor de JavaScript (que conviene recordar que es mono-hilo) esté disponible. Ahora se verá qué pasa cuando se ejecuta una función cada cierto tiempo corto. Supóngase una función relativamente larga de ejecutar (que tarde sobre 70 ms) ejecutada cada 100 ms:



## 2. Ejecución de instrucción periódicamente

En este caso como la función tarda menos tiempo que el intervalo en ejecutarse dará tiempo a lanzarla con el periodo indicado en **setInterval**. Lo que JavaScript hace es introducir en una cola de ejecución los eventos asíncronos que debe ejecutar, y lo hace en el momento en el que puede. En el ejemplo de la figura, la función se encola cada 100 ms y como tras cada ejecución aún sobra tiempo, al ir a ejecutarla nuevamente no hay nada pendiente en la cola y la ejecuta inmediatamente. Este sería el comportamiento normal deseable.

Supóngase ahora un proceso que tarda más en ejecutarse que el intervalo de repetición que se ha especificado (se puede simular mostrando desde el evento periódico un diálogo bloqueante mediante **alert** -que bloquea la ejecución hasta que se acepta- y tardando más tiempo del indicado en hacerlo). Lo que ocurre se ve reflejado en la siguiente figura:



En este caso la primera vez que se ejecuta la función se tarda más tiempo del especificado en el intervalo en terminarla. Mientras tanto el código JavaScript está detenido. Al pasar los siguientes 100 ms (el intervalo de repetición) JavaScript no puede ejecutar la función puesto que está en un bloqueo así que lo que hace es introducirla en la cola de ejecución de eventos asíncronos, a la espera de un momento libre para ejecutarla. Al acabo de otros 100 ms se debería introducir en la cola otra repetición de la tarea, pero en realidad se descarta pues sólo puede haber una de cada tipo. Finalmente, al cabo de 335 milisegundos, se termina la primera ejecución y se lanza inmediatamente la que está en la cola (que es la segunda todavía). Con lo cual el intervalo, que era de 100 ms, se ha convertido en este caso en un intervalo mucho más largo.

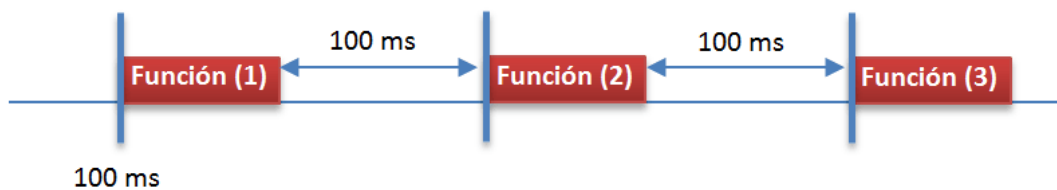
Conclusión: en condiciones extremas los intervalos no son fiables, con posibles grandes variaciones, por lo que no se puede contar siempre con que la función se ejecutará todas las veces que se pensaba. Esto puede tener importancia en ciertos tipos de aplicaciones y hay que tenerlo en cuenta.

Si realmente es importante que una tarea se repita en un intervalo de tiempo preciso tras cada ejecución (es decir, que se ejecute “x” milisegundos después de haber terminado la primera ejecución, ojo) lo que hay que hacer es emplear un temporizador simple con **setTimeout** y relanzarlo desde la propia función al final de su ejecución.

A continuación, se muestra un ejemplo de código:

```
function periodica() {  
  //Se hace lo que sea aunque tarde  
  setTimeout(periodica, 100);  
}
```

Es decir, se inicia otro nuevo intervalo al terminar la ejecución del anterior. OJO: con esto no se consiguen repeticiones cada cierto tiempo, sino que la función se repita con un periodo de tiempo predecible entre cada ejecución, que no es lo mismo, como ilustra la siguiente figura:



#### 4. Ejecución periódica de una función sin efectos colaterales



## 6.3 GENERACIÓN DE TEXTO Y ELEMENTOS HTML DESDE CÓDIGO

### 6.3.1 GENERACIÓN DE ELEMENTOS HTML DESDE CÓDIGO JAVASCRIPT

Como se ha visto a lo largo de este módulo, uno de los principales objetivos de JavaScript es convertir un documento HTML estático en una aplicación web dinámica. De esta forma, es posible manipular los objetos que representan el contenido de una página web con el fin de crear documentos dinámicos.

Por ejemplo, se podría informar en el encabezado del idioma del navegador:

```
let lang = navigator.language;  
document.write("<h1>El idioma del navegador utilizado es: " + lang + "</h1>");
```

Posteriormente se verán más ejemplos de generación de elementos HTML, especialmente en el apartado de ventanas.

### 6.3.2 LISTAS EN REACT

En cuanto a React, para ilustrar la generación de elementos HTML desde código, en el ejemplo a continuación en el que se va a ver cómo poder reutilizar un componente en múltiples elementos. En primer lugar, se va a crear un componente Ficha. Su misión es mostrar una ficha de una foto en una galería.

El componente Ficha (guardado como components/Ficha.js) tendrá el siguiente código:

```
function Ficha(props) {  
  const foto = props.foto;  
  return (  
    <div className="ficha" id={foto.id}>  
      <h2>{foto.titulo}</h2>  
      <p>{foto.descripcion}</p>  
      <figure className="elemento-foto">  
        <img src={` /images/${foto.url}`} alt={foto.alt} />  
      </figure>  
    </div>  
  )  
}  
export default Ficha;
```

Este componente recibe la foto como **prop** y renderiza sus distintas propiedades como el título, la descripción, la propia foto a partir de una URL y su texto alternativo.

Por otra parte, se va a utilizar una clase Foto que contenga las propiedades que debe tener cada una de las imágenes renderizadas.

Se va a crear un directorio llamado **data** donde ubicar esta clase y la colección de fotos que se renderizará finalmente.

```
class Foto {  
  constructor(inicializador) {  
    this.id = inicializador.id;  
    this.titulo = inicializador.titulo;  
    this.url = inicializador.url;  
    this.alt = inicializador.alt;  
    this.descripcion = inicializador.descripcion;  
    this.fecha = new Date();  
  }  
  esNueva() {  
    return this.id === undefined;  
  }  
}  
export default Foto;
```

Esta clase se usará en un array que va a contener las distintos objetos Foto a ubicar en la galería. El archivo se llamaría **ColFotos.js**

```
import Foto from "./Foto.js";  
export const ColFotos = [  
  new Foto({  
    id: "001",  
    titulo: "Imagen 1",  
    url: "imagen1.png",  
    alt: "Imagen 1",  
    descripcion: "Esta es la imagen número 1",  
    fecha: new Date(),  
  }),  
  new Foto({  
    id: "002",  
    titulo: "Imagen 2",  
    url: "imagen2.png",  
    alt: "Lobo",  
    descripcion: "Esta es la imagen número 2",  
    fecha: new Date(),  
  }),  
  new Foto({  
    id: "003",  
    titulo: "Imagen 3",  
    url: "imagen3.png",  
    alt: "Luna",  
    descripcion: "Lorem iincsoluta.",  
    fecha: new Date(),  
  }),  
];
```

Por último, el componente **Galeria.js** renderiza la colección de fotos.

```
import Ficha from "../Ficha";
function Galeria(props) {
  const fotos = props.fotos;
  return (
    <div className="galeria">
      {fotos.map((foto) => (
        <Ficha key={foto.id} foto={foto} />
      ))}
    </div>
  );
}
export default Galeria;
```

Este componente recoge como propiedad la colección de fotos enviada al constructor mediante props. Internamente, el componente se constituye con un elemento **Ficha**, con dos atributos: **key**, que utiliza React para identificar al componente y **foto** con el objeto con sus datos. Lo interesante en este caso es el uso de **map**, muy extendido en React, para recorrer una colección. Ya se vio anteriormente el funcionamiento junto con **reduce** mostrando el modelo MapReduce. Finalmente, **App.js** quedaría así:

```
import Galeria from "../components/Galeria";
import {ColFotos} from "../data/ColFotos";
function App() {
  return (
    <div className="App">
      <Galeria fotos={ColFotos} />
    </div>
  );
}
```

## 6.4 CREACIÓN DE NUEVAS VENTANAS. COMUNICACIÓN ENTRE VENTANAS

### 6.4.1 GESTIÓN DE VENTANAS EN JAVASCRIPT

JavaScript permite gestionar diferentes aspectos relacionados con las ventanas como por ejemplo abrir nuevas ventanas al presionar un botón. Cada una de estas ventanas tiene un tamaño, posición y estilo diferente. Estas ventanas emergentes suelen tener un contenido dinámico. Además, se puede crear una comunicación e interacción entre dos o más ventanas.

Abrir y cerrar nuevas ventanas es una operación muy común en las aplicaciones web incluso sin la intervención del usuario. HTML permite abrir nuevas ventanas, pero no permite ningún control posterior sobre ellas. Con JavaScript es posible abrir una ventana vacía mediante el método **open()**:

```
let nuevaVentana = window.open();
```

De este modo la variable llamada nuevaVentana contendrá una referencia a la ventana creada pudiéndose utilizar posteriormente para cualquier manipulación que se quiera hacer sobre la ventana mientras se ejecuta JavaScript. El método **open()** cuenta con tres argumentos:

- URL.
- Nombre de la ventana.
- Colección de atributos que definen la apariencia de la ventana.

Ejemplo:

```
let nuevaVentana = window.open('http://www.misitioWeb.com', 'Publicidad',  
'height=100, width=100');
```

Para cerrar una ventana se puede invocar el método **close()**. Por ejemplo, introduciendo un botón que permita cerrar dicha ventana:

```
<button onClick="window.close()">Cerrar</button>
```

En cuanto a la comunicación entre ellas, desde una ventana se pueden abrir o cerrar otras nuevas. La primera se denomina ventana principal, mientras que las segundas se denominan ventanas secundarias. Desde la ventana principal se puede acceder a las ventanas secundarias.

En el siguiente ejemplo se ilustra cómo una ventana principal se comunica con una emergente (**popup.html**). Nota: por la naturaleza del ejemplo, probablemente no funcione correctamente con servidores Live como el de Visual Studio Code. Se puede probar accediendo directamente al fichero .html de la página principal.

```
<html Lang="es">
  <body>
    <h1>Ventana Principal</h1>
    <button id="abrePopup">Abrir Ventana</button>
    <button id="enviaMensaje">Enviar Mensaje</button>
    <script>
      let popup;

      // Se abre ventana emergente
      document.getElementById("abrePopup").addEventListener("click", () => {
        popup = window.open("popup.html", "Popup", "width=400,height=300");
      });

      // Se envía un mensaje a la ventana
      document.getElementById("enviaMensaje").addEventListener("click", () => {
        if (popup && !popup.closed) {
          popup.postMessage("Te saludo desde la ventana principal", "*");
        } else {
          alert("La ventana emergente no está abierta.");
        }
      });
    </script>
  </body>
</html>
```

Y el archivo popup.html podría ser como este:

```
<!DOCTYPE html>
<html Lang="es">
  <body>
    <h1>Ventana Emergente</h1>
    <p id="mensajeRecibido">Esperando mensaje...</p>

    <script>
      // Se escucha un mensaje de la ventana principal
      window.addEventListener("message", (event) => {
        // Se muestra el mensaje recibido
        document.getElementById(
          "mensajeRecibido"
        ).textContent = `Mensaje recibido: ${event.data}`;
      });
    </script>
  </body>
</html>
```

En este caso, **addEventListener** escucha un tipo de evento "message" el cual se dispara cuando la ventana recibe un mensaje a través de la API `postMessage`.

En cualquier caso, esta forma de comunicación entre ventanas no es la más apropiada y elegante. Es más apropiado, por ejemplo, usar almacenamiento compartido con **LocalStorage** (ya visto), **SessionStorage** u opciones más modernas como [Broadcast Channel API](#).

### 6.4.2 REACT ROUTER

Dentro de la gestión de ventanas, en este apartado se va a ver cómo se pueden crear y gestionar varias páginas y/o ventanas con React. Para ello, se presenta una biblioteca, **React router**, la cual permite crear rutas en una aplicación React. Lo primero que hay que hacer para utilizarla es instalar la dependencia como se ha hecho en otras ocasiones:

```
npm i react-router-dom
```

Una vez instalada, debería de aparecer en el archivo **package.json** tal como se puede ver en la imagen (dentro de la sección dependencies):

```
"dependencies": {  
  "@testing-library/jest-dom": "^5.17.0",  
  "@testing-library/react": "^13.4.0",  
  "@testing-library/user-event": "^13.5.0",  
  "react": "^18.3.1",  
  "react-dom": "^18.3.1",  
  "react-router-dom": "^7.1.4",  
  "react-scripts": "^5.0.1",  
  "web-vitals": "^2.1.4"  
},
```

5. React router en package.json

Para ilustrar el uso de React router y la utilización de rutas, se van a crear una serie de páginas en el proyecto las cuales se podrían ubicar en un directorio **src/pages** que habrá que dar de alta en el directorio donde esté la aplicación. Al fin y al cabo, una página será en realidad un componente. En primer lugar, se crea la página Home (**Home.js**) con el siguiente código:

```
import { Link } from "react-router-dom";  
const Home = () => {  
  return (  
    <section id="home">  
      <h1>Bienvenid@</h1>  
      <Link to="/pagina1">Entrar</Link>  
    </section>  
  );  
};  
export default Home;
```

Lo primero que hace este código es importar el elemento **Link** que apunta a otra página (**pagina1**) además de contener un elemento sección. No tiene nada más y aquí lo interesante es ver el componente como página y cómo se va a utilizar como ruta posteriormente.

A continuación, se diseña una segunda página, `Pagina1` (`Pagina1.js`) tal que así:

```
const Pagina1 = () => {  
  return <>Página 1</>;  
};  
export default Pagina1;
```

No tiene nada de particular ya que solo sirve para ilustrar el uso de rutas; en definitiva, muestra un texto nada más.

Para definir cómo será la aplicación web que englobe a estas páginas y otras posibles, hay que acudir al `App.js` y darle esa estructura:

```
import { Route, Routes } from "react-router-dom";  
import Home from "../pages/Home";  
import Pagina1 from "../pages/Pagina1";  
function App() {  
  return (  
    <Routes>  
      <Route path="/" element={<Home />} />  
      <Route path="/pagina1" element={<Pagina1 />} />  
    </Routes>  
  );  
}  
export default App;
```

En este caso, se importa un elemento **Routes**, que contendrá el mapa de rutas y cada una de las páginas se incluirá dentro de él como un elemento **Route**. Cada elemento **Route** consta de la ruta ajustada mediante el atributo **path** y en **element**, se indica el elemento que representa a cada página al igual que se ha estado viendo hasta ahora con cada componente. No hay que olvidar que estos elementos deben incluirse entre llaves al ser un código a ejecutar.

Aunque el trabajo parece concluido, falta hacer un ajuste en el propio fichero raíz de la aplicación (`index.js`) y es que, en estas arquitecturas hay que incluir un elemento **BrowserRouter** que contenga el componente `App` para que así funcione. Por tanto, `root.render` se llamará de este modo:

```
root.render(  
  <React.StrictMode>  
    <BrowserRouter>  
      <App />  
    </BrowserRouter>  
  </React.StrictMode>  
);
```

Si ahora se prueba la aplicación, se entrará directamente en el home y se podrá navegar mediante el botón entrar a la página 1.

## Parámetros

En el ejemplo anterior se vio un sistema básico de navegación, pero lo interesante no suele ser ir a páginas concretas sin más; lo más común es acceder a vistas personalizadas a partir de un parámetro.

Dado el ejemplo anterior, supóngase que se tiene una lista de productos y se quiere consultar el que tiene un id concreto. En el ejemplo, se verá de forma superficial para ilustrar de forma sencilla el uso del paso de parámetros, pero lo suyo sería que, a partir del identificador dado, se obtenga más información del producto llamando a una API, por ejemplo. Ahora se creará una nueva página llamada Producto (Producto.js) como la siguiente:

```
import { useParams } from "react-router-dom";

const Producto = () => {
  const { id } = useParams();
  return (
    <section id="producto">
      Estás consultando el producto con identificador {id}
    </section>
  );
};

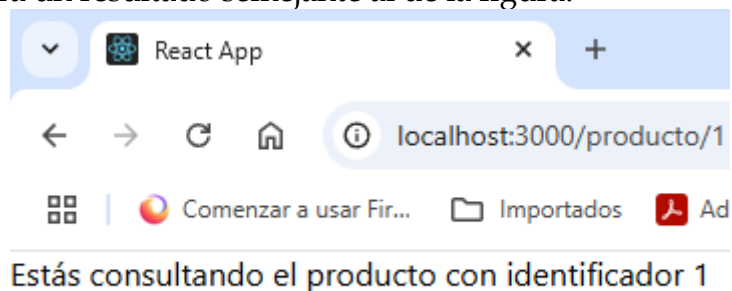
export default Producto;
```

¿Qué hace esta página? Pues sencillamente recoge el identificador mediante el hook useParams y a partir de ahí renderiza un contenido, que en este caso es bien sencillo, un texto con el identificador.

Ahora, para poder usar esta ruta: [http://ip/producto/\[id\]](http://ip/producto/[id]) hay que definirla en la lista de rutas contenida en App.js y se hace añadiendo la siguiente línea:

```
<Route path="/producto/:id" element={<Producto />} />
```

El formato es muy simple: ruta/:id y se enlaza al propio componente/página Producto. Si se llama a la URL se verá un resultado semejante al de la figura.



### 6. Uso de parámetros en rutas

Ahora sería relativamente sencillo modificar la página Home para poder seleccionar el número de id, por ejemplo, de un desplegable y abrir la página con ese parámetro.



### 6.4.3 ENTENDIENDO EL RENDERIZADO REACT

Dos de los temas centrales cuando se trata de entender cómo se renderiza React son la memorización y las matrices de dependencias. La memorización es una piedra angular en la optimización de React; se trata de una técnica centrada en la renderización eficiente y en evitar recálculos innecesarios. El uso efectivo de los hooks **memo()**, **useMemo** y **useCallback** de React puede mejorar significativamente el rendimiento de la aplicación.

En el núcleo de la naturaleza reactiva de React se encuentran los arrays de dependencia que se usan en hooks como **useEffect** y **useMemo**. Estos arrays son cruciales para determinar cuándo y cómo se actualizan los componentes. Un componente funcional se renderizará por una de estas tres razones:

- El componente acaba de ser montado (antes no estaba en el árbol de componentes y ahora sí).
- El componente padre se ha vuelto a renderizar.
- El componente usa un hook que ha marcado este componente para ser vuelto a renderizar.

Si ninguna de estas cosas ocurre, el componente no se re-renderizará, lo que es una garantía. Si cualquiera de las tres ocurre, el componente se volverá a renderizar seguro, pero React podría hacer un renderizado en lote si ocurren varias de estas cosas. Si un valor de estado cambia y el componente padre vuelve a renderizarse, por ejemplo, el componente se podría volver a renderizar una vez e incluso dos veces. Este proceso se controla por React y depende de detalles de tiempo muy sutiles. Hay una creencia muy común sobre React que afirma que los componentes se re-renderizan porque sus propiedades cambian, pero no es así, como se puede probar de dos formas distintas:

- Se puede crear un componente que tenga propiedades que cambien, pero no se vuelva a renderizar
- Se puede crear un componente que se renderice muchas veces con las mismas propiedades y se re-renderice cada vez.

Ambas situaciones se pueden imaginar fácilmente. Primeramente, se necesita un componente que claramente muestre cómo se re-renderiza.

```
import { useEffect } from "react";
import { useRef } from "react";
const Rerenderizable = () => {
  const primeraVez = useRef(true);
  useEffect(() => {
    primeraVez.current = false;
  }, []);
  const style = { color: primeraVez.current ? "red" : "blue" };
  const text = primeraVez.current
    ? "Este es el primer renderizado"
    : "Este no es el primer renderizado";
  return <p style={style}>{text}</p>;
};
export default Rerenderizable;
```

Este código tiene un comportamiento muy particular. Si se renderiza por primera vez, aparece el mensaje correspondiente. Aunque se refresque la página, seguirá mostrándolo porque no deja de ser un “primer” renderizado. La diferencia viene si se modifica algo y se produce el comportamiento reactivo, es decir, la página se re-renderiza; entonces mostrará el mensaje “Este no es el primer renderizado”. Obviamente, tal como está articulado el componente, la única forma de volver a renderizarlo sin agentes externos es hacer una mínima modificación en el código y guardarla; en ese caso, ya se mostrará el mensaje en azul. La cuestión de fondo es que se cambia primeraVez y el componente no se re-renderiza como se cree habitualmente y que resulta ser un mito.

Por otra parte, este código usa dos hooks interesantes a la hora de trabajar en React:

- **useEffect**. Este es uno de los hooks más utilizados y ya se ha utilizado en ejemplos anteriores. Se usa para ejecutar lógica adicional una vez que React haya representado completamente el componente. Su estructura básica es la siguiente:

```
useEffect(() => {  
  // Código que se ejecuta como efecto  
  return () => {  
    // Código opcional de limpieza (cleanup)  
  };  
}, [dependencias]);
```

El código que se ejecuta como efecto puede contener llamadas a APIs, actualización del DOM, cambios en valores externos, etc.

En cuanto a la lista de dependencias, se trata de una lista de variables o valores que React debe observar. Este array indica cuándo debe ejecutarse el efecto y según su contenido, el efecto se comportará de distintas maneras. Por ejemplo, si el array está vacío como en el ejemplo, el efecto se ejecuta solo una vez después del primer renderizado del componente. En definitiva, aquí lo que se persigue es que el código del primer parámetro se ejecute una sola vez (por ejemplo, inicializar un valor o llamar a una API). Si se incluyen variables en el array de dependencias, el efecto se ejecutará cada vez que alguna de esas dependencias cambie. React compara los valores actuales de las dependencias con sus valores anteriores y decide si debe o no volver a ejecutar el efecto. Por ejemplo:

```
useEffect(() => {  
  console.log("El valor de contador cambió a:", contador);  
}, [contador]);
```

Es muy útil si se quiere responder a cambios de estado, de props o de cualquier otra variable externa que afecte al componente.

- **useRef**. Este hook permite guardar un valor persistente que no se reinicia con cada renderizado del componente. En este caso, se asigna a la constante **primeraVez** como true para indicar que es la primera vez que se representa el componente. Si se cambiara el valor de **primeraVez.current** no forzaría un nuevo renderizado del componente. La propiedad **.current** pertenece al objeto **useRef** y almacena el valor actual de esa referencia.

A continuación, un ejemplo de re-renderizado que involucra a dos componentes. En él se van a desencadenar re-renderizados para intentar desmontar el mito visto anteriormente. Un primer componente `RerenderizadorSinCambioProps.js`:

```
import { useState } from "react";
import Rerenderizable from "../Rerenderizable";
const RerenderizadorSinCambioProps = () => {
  const [, setContador] = useState(0); // #1
  return (
    <div>
      <button onClick={() => setContador((c) => c + 1)} /* #2 */>
        Click para re-renderizar
      </button>
      <Rerenderizable /> /* #3 */
    </div>
  );
};
export default RerenderizadorSinCambioProps;
```

Un segundo componente `NoRenderizadorConCambioProps.js`:

```
import { useRef } from "react";
import Rerenderizable from "../Rerenderizable";
const NoRenderizadorConCambioProps = () => {
  const contador = useRef(0); // #4
  return (
    <div>
      <button onClick={() => (contador.current++)} /* #5 */>
        Click para re-renderizar
      </button>
      <Rerenderizable contador={contador.current} /> /* #6 */
    </div>
  );
};
export default NoRenderizadorConCambioProps;
```

Y el `App.js` quedaría así:

```
import RerenderizadorSinCambioProps from "../components/RerenderizadorSinCambioProps";
import NoRenderizadorConCambioProps from "../components/NoRenderizadorConCambioProps";
function App() {
  return (
    <main>
      <h4>Re-renderizado sin cambiar propiedades</h4>
      <RerenderizadorSinCambioProps />
      <h4>No re-renderiza con cambio de propiedades</h4>
      <NoRenderizadorConCambioProps />
    </main>
  );
}
export default App;
```

Se explican a continuación cada una de las marcas en los comentarios:

- #1 Inicializa el estado, pero se necesita actualizar, no leer.
- #2 Actualiza el estado para forzar un re-renderizado.
- #3 Incrusta el componente Rerenderizable sin propiedades.
- #4 Define una referencia, que se puede actualizar sin disparar un re-renderizado
- #5 Actualiza el valor dentro de la ref
- #6 Incluye el componente Rerenderizable con una propiedad extraída de la referencia ref.

Si se prueba esta aplicación, se verá que, al pulsar el primer botón, el mensaje cambia. Se ha incrementado una variable de estado contador en el control padre pero no se cambia nada en el hijo (Rerenderizable). Al cambiar el estado del padre, el hijo se renderiza de nuevo, con lo que lanza el mensaje correspondiente.

En el segundo botón, no se cambia el estado del padre, sencillamente se usa la referencia contador la cual se modifica al pulsar el botón incrementándola en 1. Como ya se vio, useRef guarda un valor persistente que no se reinicializa con cada re-renderizado. La diferencia en este caso es que se manda el valor al hijo Rerenderizable mediante la prop contador. Obviamente, no se vuelve a renderizar porque el hecho de pasarle una propiedad no desencadena dicha acción y el propio padre tampoco al no cambiar más que una variable de referencia.

Lo interesante de esta aplicación es que, si el mito fuera cierto, el componente Rerenderizable dentro de RerenderizarConCambioProps no se volvería a renderizar nunca porque las propiedades tampoco cambian. Del mismo modo, el componente Rerenderizable dentro de NoRenderizarConCambioProps debería volverse a renderizar incluso aunque el padre no lo haga ya que la propiedad pasada ha cambiado. Y en la realidad, no pasa ninguna de estas cosas como comprobarse en la ejecución.

### **Re-renderizado sin cambiar propiedades**

Click para re-renderizar

Este no es el primer renderizado

### **No re-renderiza con cambio de propiedades**

Click para rerenderizar

Este es el primer renderizado

#### **7. Ejemplo de re-renderizado**

En conclusión, esto significa que los componentes React solo se vuelven a renderizar cuando lo hacen sus padres independientemente de qué propiedades tomen, de donde vengan esas propiedades o cómo se actualicen potencialmente.

## 6.5 GESTIÓN DE LA APARIENCIA DE LA VENTANA

### 6.5.1 APARIENCIA DE LAS VENTANAS

El método **window.open()** cuenta con propiedades que permiten decidir su tamaño, ubicación o los elementos que contendrá:

- **height**: Corresponde a la altura de la ventana.
- **left**: Corresponde a la distancia en píxeles desde el lado izquierdo del área de trabajo definida por el sistema operativo del usuario donde se generará la nueva ventana.
- **menubar**: Determina si va a ser visible o no la barra de menús.
- **resizable**: Determina si se va a poder cambiar o no el tamaño de ventana.
- **scrollbars**: Determina si van a ser visibles o no las barras de desplazamiento.
- **status**: Determina si va a ser visible o no la barra de estado.
- **toolbar**: Determina si va a ser visible o no la barra de herramientas.
- **top**: Corresponde a la distancia en píxeles desde la parte superior del área de trabajo definida por el sistema operativo del usuario donde se generará la nueva ventana
- **width**: Corresponde a la anchura de la ventana.

Los argumentos que definen la altura y la anchura se establecen determinando el tamaño en píxeles. Los demás argumentos sirven para decidir si mostrar o no un elemento de la ventana, así que los posibles valores que pueden tomar estos últimos son uno (1) o cero (0). A continuación, se puede ver un ejemplo de llamada a open:

```
window.open('', 'Ventana', 'top=20,left=20,width=250,height=250');
```

### 6.5.2 ESTILOS EN REACT

En cuanto a React, la gestión de la apariencia se puede realizar aplicando estilos CSS. Esto se puede hacer de varias formas, entre ellas, estilos en línea, módulos CSS, componentes con estilo, etc. Se verá cada aproximación a continuación.

#### Estilos en línea

En el siguiente código se ilustra cómo aplicar estos estilos en línea:

```
import React from "react";
function MiComponente() {
  return (
    <div style={{
      backgroundColor: "lightblue",
      padding: "20px",
      borderRadius: "5px",
    }}>
      <p style={{ color: "white", fontSize: "18px" }}>Esto es un párrafo con estilos
en línea</p>
    </div>);
}
export default MiComponente;
```

En este código el elemento `<div>` tiene estilos en línea aplicados directamente mediante el atributo `style`. Del mismo modo, como puede verse, se aplican en el párrafo. En definitiva, consiste en aplicar el estilo ad-hoc a cada elemento.

## Módulos CSS

Un fichero CSS se puede cargar como si fuera un módulo en React pero para ello debe cumplir una condición: tener el mismo nombre que el componente añadiendo el sufijo `.module`. Para este ejemplo, en primer lugar, se crea un fichero llamado `Alternamensaje.module.css`:

```
.mensaje {  
  display: block;  
  color: green;  
  font-size: 18px;  
  margin-top: 10px;  
}  
.oculto {  
  display: none;  
}
```

A continuación, se incluye el código propio del componente. Obsérvese cómo importa los estilos en la siguiente línea:

```
import estilos from "../Alternamensaje.module.css";
```

El código completo es el siguiente:

```
import React, { useState } from "react";  
import estilos from "../Alternamensaje.module.css";  
  
function Alternamensaje() {  
  const [esVisible, setEsVisible] = useState(true);  
  const alternaVisibilidad = () => {  
    setEsVisible(!esVisible);  
  };  
  return (  
    <div>  
      <h2>Alternamensaje</h2>  
      <button onClick={alternaVisibilidad}>  
        {esVisible ? "Esconde mensaje" : "Muestra mensaje"}  
      </button>  
      <p className={esVisible ? estilos.mensaje : estilos.oculto}>  
        Este es un mensaje oculto  
      </p>  
    </div>  
  );  
}  
  
export default Alternamensaje;
```

Como ya se citó, lo primero que hace el código es importar el módulo CSS como un objeto llamado `estilos`. Ya se habían visto anteriormente los módulos JavaScript, pero no se había indicado que también se pueden importar objetos de módulos CSS. Las reglas de este CSS son de ámbito local para prevenir conflictos entre componentes.

Además, se inicializa una variable de estado `esVisible` que representa cuándo el párrafo del mensaje estará visible o no. El método `alternaVisibilidad` cambia la visibilidad del mensaje modificando la variable de estado cuando se hace clic sobre el botón. Por último, el párrafo será visible o no si se aplica un estilo u otro en función de la variable de estado.

### Componentes con estilo

CSS también se puede aplicar en componentes React mediante objetos JavaScript que en el siguiente código se nombra como `estiloMensaje`.

Por ejemplo, en el código siguiente hay un objeto con **nombre** `mensajeEstilo` { `color: 'green', fontSize: '18px'` } representa propiedades para el color y tamaño de letra. Esto también tiene un estilo condicional. Por ejemplo, { `display: esVisible ? 'block' : 'none'` } que dinámicamente ajusta la propiedad `display` basada en el valor de `esVisible`. El código es el siguiente:

```
import React, { useState } from "react";

const AlternaMensaje = () => {
  const [esVisible, setEsVisible] = useState(true);
  const alternaVisibilidad = () => {
    setEsVisible(!esVisible);
  };
  const estiloMensaje = {
    display: esVisible ? "block" : "none",
    color: "green",
    fontSize: "18px",
    marginTop: "10px",
  };
  return (
    <div>
      <h2>Alterna Mensaje</h2>
      <button onClick={alternaVisibilidad}>
        {esVisible ? "Esconde mensaje" : "Muestra mensaje"}
      </button>
      <p style={estiloMensaje}>Este es un mensaje oculto</p>
    </div>
  );
};

export default AlternaMensaje;
```



### 6.5.3 FRAMEWORKS DE ESTILOS EN REACT: MATERIAL-UI

Existen multitud de bibliotecas de componentes que pueden ahorrar mucho trabajo en el diseño de una aplicación web y de la apariencia de las distintas páginas y/o componentes. Anteriormente se vio cómo integrar Font Awesome, el cual permite utilizar fuentes e iconos mediante CSS. En este apartado se verá otra opción: [Material-UI](#), biblioteca de componentes diseñada por Google. El estilo de estos componentes se basa en el concepto Material, reconocible por su aspecto minimalista, uso de estilo plano en componentes y botones además de convergencia general tanto en dispositivos móviles como en aplicaciones web. Material se puede aplicar a todo tipo de frameworks y bibliotecas incluyendo, por supuesto, a React. Para ello, hay que descargarlo y agregarlo mediante el gestor de paquetes al igual que se ha hecho otras veces:

```
npm i --save @mui/material @mui/icons-material @emotion/react @emotion/styled
```

Para probar Material, primero se va a codificar una aplicación principal que usará componentes Material además de uno personalizado que se verá después.

```
import './App.css';
import { useState } from 'react';
import Box from '@mui/material/Box'
import Tab from '@mui/material/Tab'
import Tabs from '@mui/material/Tabs'
import MaterialForm from './components/MaterialForm';
function App() {
  const [tabVisible, setTabVisible] = useState(0);
  const manejador = (evento, indice) => {
    console.log(indice);
    setTabVisible(indice);
  }

  return (
    <div>
      <Box sx={{borderBottom: 1, borderColor: 'divider'}}>
        <Tabs value={tabVisible} onChange={manejador} aria-label='Ejemplo básico de tabs'>
          <Tab label='Tab 1' />
          <Tab label='Tab 2' />
          <Tab label='Tab 3' />
        </Tabs>
      </Box>
      {tabVisible === 0 && <div>Item 1</div>}
      {tabVisible === 1 && <div>Item 2</div>}
      {tabVisible === 2 && <div>Item 3</div>}
      <MaterialForm />
    </div>
  );
}
export default App;
```



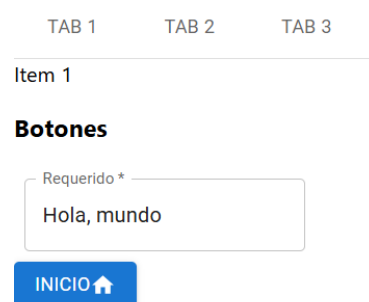
Como puede verse se usan elementos como un conjunto de pestañas o tabs integrados en una caja con un separador. Este conjunto de pestañas tiene asociado un manejador para el cambio de pestañas. Fuera de esa caja, en función de qué **Tab** esté visible (manejado por una variable de estado), se muestra un contenido u otro.

Por otra parte, se incluye un componente `MaterialForm.js` que consta de una caja estilo formulario que contiene un cuadro de texto y un botón con un icono "Home". Uno de los atributos de **Box** que más llama la atención es `sx`, el cual permite aplicar estilos directamente sobre el componente.

El código de `MaterialForm.js` quedaría así:

```
import Button from '@mui/material/Button'
import Box from '@mui/material/Box'
import TextField from '@mui/material/TextField'
import HomeIcon from '@mui/icons-material/Home'
const MaterialForm = () => {
  return (
    <>
      <h3>Botones</h3>
      <Box component='form' noValidate autoComplete='off'
        sx = {{ '& .MuiTextField-root': {m: 1, width: '25ch'}, }}>
        <div>
          <TextField required id='outlined-required' label="Required"
defaultValue="Hola, mundo" />
        </div>
        <Button variant='contained'>
          Inicio<HomeIcon fontSize='small' />
        </Button>
      </Box>
    </>
  )
}
export default MaterialForm;
```

El resultado sería similar al siguiente:



TAB 1   TAB 2   TAB 3

Item 1

**Botones**

Requerido \*

Hola, mundo

INICIO 🏠

## 8. Ejemplo de interfaz Material-UI

## 6.6 INTERACCIÓN CON EL USUARIO

La interacción con el usuario implica capturar eventos generados por acciones del usuario (clics, pulsación de una tecla, acción de desplazamiento, etc.) y responder a ellos mediante cambios visuales o funcionales en la interfaz.

Ya se ha visto en anteriores unidades cómo trabajar la gestión de eventos en JavaScript con **addEventListener** y se volverá a tratar en unidades posteriores ya que es una de las claves del desarrollo web.

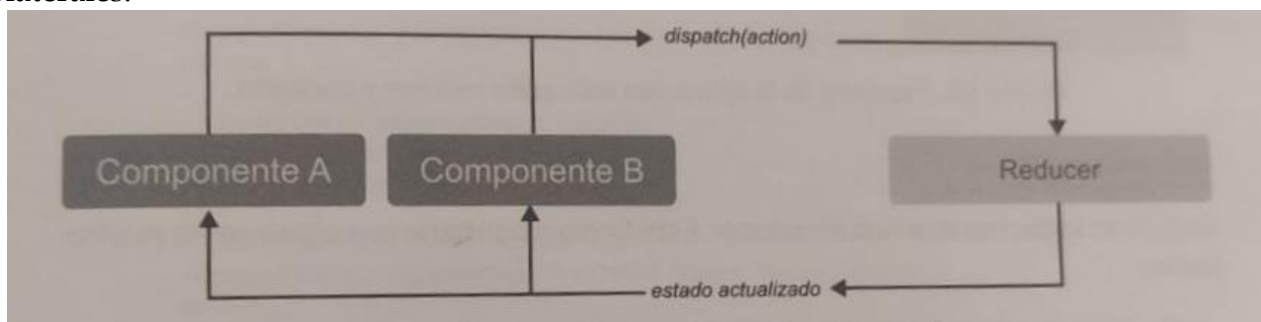
Se puede decir que en React, el enfoque de interacción está basado en los componentes y en su estado. También se ha visto anteriormente cómo se declaran y programan manejadores de evento a través de propiedades JSX como **onClick**, **onChange**, etc. En realidad, coinciden con los eventos HTML, pero cambiando a mayúscula la letra propia del evento (Click, Change, ...).

En cuanto a la gestión de estado, React proporciona un mecanismo conocido como Reducer en el que se profundiza a continuación.

### GESTIÓN DE ESTADO CON REDUCERS

La gestión de estado general de una aplicación React es crucial. Ya se ha sugerido anteriormente que lo deseable en React es usar funciones puras siempre que sea posible y aislar los cambios de estado para que se hagan solo en un sitio. Para ayudar a esto, se pueden usar los reducers, que no son más que funciones que se encargan de todos los cambios de estado.

Los componentes interactúan con el estado mandando acciones al reducer. Este se encarga de procesar esta acción y genera el nuevo estado. De esta forma, se crea una especie de estado global para toda la aplicación cuyos cambios se canalizan por un solo punto evitando efectos colaterales.



9. Esquema de funcionamiento de un reducer

React permite definir una función reducer de forma manual que luego se puede usar en los componentes mediante el hook **useReducer**. Un **reducer** toma dos argumentos: el estado actual de un componente y una acción. Dependiendo de la acción dada, el **reducer** devuelve un nuevo estado. En el siguiente ejemplo, el **reducer** manejará el estado de un contador con tres operaciones: incremento, decremento y reinicio.

Lo primero que hay que hacer es definir esa función **reducer** con sus argumentos y lo elegante es hacerlo en un fichero independiente, por ejemplo, **contadorReducer.js** el cual tendría un código como este:

```
const contadorReducer = (estado, accion) => {  
  switch (accion.tipo) {  
    case 'INCREMENTO':  
      return {contador: estado.contador + 1};  
    case 'DECREMENTO':  
      return {contador: estado.contador - 1};  
    case 'REINICIO':  
      return { contador: 0 };  
    default:  
      return estado;  
  }  
};  
export default contadorReducer;
```

Una vez definido el reducer, es momento de utilizarlo en un componente con `useReducer`. Este hook toma dos argumentos: el reducer y el estado inicial. A partir de esos datos, lo que hace es devolver un array con el estado actual y una función `dispatch` que se utilizará para enviar acciones al reducer.

```
import React, { useReducer } from 'react';  
import contadorReducer from '../contadorReducer';  
const estadoInicial = { contador: 0 };  
const Contador = () => {  
  const [estado, dispatch] = useReducer(contadorReducer, estadoInicial);  
  return (  
    <div>  
      <h1>Contador: {estado.contador}</h1>  
      <button onClick={() => dispatch({ tipo: 'INCREMENTO' })}>Incrementar</button>  
      <button onClick={() => dispatch({ tipo: 'DECREMENTO' })}>Decrementar</button>  
      <button onClick={() => dispatch({ tipo: 'REINICIO' })}>Resetear</button>  
    </div>  
  );  
};  
export default Contador;
```

Y ya solo quedaría integrar el componente en la aplicación o en otro componente. La función `contadorReducer` maneja las acciones que se le envían. Dependiendo del tipo de acción (INCREMENTO, DECREMENTO, REINICIO), modifica el estado del contador. En el componente `Contador` se utiliza `useReducer` para inicializar el estado del contador. `estado` contiene el estado actual, y `accion` es la función que se usa para enviar esas acciones al reducer. Cada botón tiene un `onClick` que llama a `accion` con la específica para cada operación. Cuando se hace clic en un botón, se envía la acción al reducer, que actualiza el estado según la solicitud.

La diferencia con otros casos vistos hasta ahora es que el reducer permite ser utilizado desde varios componentes distintos centralizando las acciones a realizar.

## 6.7 MECANISMOS DEL NAVEGADOR PARA EL ALMACENAMIENTO Y RECUPERACIÓN DE INFORMACIÓN

### 6.7.1 COOKIES

Las cookies son unos pequeños ficheros de texto (hasta de 4Kb) que las aplicaciones web almacenan en el navegador del usuario con el objetivo de recuperar en futuras sesiones los datos que almacenan. Esto puede ser necesario por la propia naturaleza del protocolo HTTP, el cual no tiene estado. Como ya se vio anteriormente, objetos como **LocalStorage** permiten un almacenamiento local, y en esa línea también están las cookies.

Si bien es cierto que esta posibilidad ha abierto la puerta a usos abusivos e incluso maliciosos, ya existe una fuerte regulación europea que trata de controlar su uso.

El objeto que proporciona JavaScript para trabajar con cookies es **document.cookie** con el que pueden definirse y obtenerse cookies propias (un máximo de 20 por dominio).

Para crear una cookie y guardarla en el navegador del usuario simplemente hay que asignar a la propiedad **document.cookie** una cadena de caracteres con la forma **clave=valor**. Por ejemplo:

```
document.cookie = "idioma=es";  
document.cookie = "esquemaColor=dark";  
document.cookie = "haVotadoEncuesta=si";
```

A medida que se añade una nueva clave con su valor, la propiedad **document.cookie** la irá anexando a las anteriores separándola con un punto y coma (;).

Otra utilidad interesante de las cookies es que se les puede indicar un horizonte de vida, es decir, una fecha a partir de la cual la cookie se destruye. Técnicamente, cada vez que el usuario cierra el navegador, las cookies se liberan, pero si se desea que persista hasta una fecha indicada, se debe incluir un valor. Este puede ser expresado como **max-age** con un número de segundos (a partir del momento de creación de la cookie) o **expires** con una fecha en formato GMT:

```
let limiteSegundos = 60 * 60 * 24 * 365; // Un año  
document.cookie = `esquemaColor=dark; max-age=${limiteSegundos}`;
```

También es importante destacar que las cookies solo son accesibles por aplicaciones del mismo dominio donde se crearon y siempre que estén en el mismo directorio. Si la página en la que se grabó la cookie es `midominio.com/home` cualquier página fuera de esta ruta no podrá leerla. Sin embargo, se puede indicar desde qué ruta del dominio se quiere que se pueda acceder a la cookie expresando la ruta como valor de la clave **path**:

```
document.cookie = "haVotadoEncuesta=si; path=/";
```

Por último, para borrar una cookie es suficiente con indicar como fecha de expiración cualquier fecha del pasado.

La lectura de una cookie se haría accediendo a esa propiedad, **document.cookie**, pero claro, hay que evaluar la cadena completa y es tarea de desarrollo diseñar el código para leerla. Una opción sencilla ayudándose de las expresiones lambda podría ser la siguiente:

```
const getValorCookie = (name) =>
  document.cookie
    .split("; ")
    .find((row) => row.startsWith(name + "="))?
    .split("=")[1];
```

La interrogación después del find pertenece a lo que se conoce como encadenamiento opcional la cual asegura que el código no provoque un error si el objeto al que intenta acceder es **null** o **undefined**. Así se garantiza que el código no lance un error al ejecutar el **split**.

### 6.7.2 OTROS MECANISMOS DE ALMACENAMIENTO LOCAL

En HTML5 se introdujeron dos nuevos objetos para el almacenamiento de datos en el lado del cliente: **sessionStorage** y **localStorage**. Una de las principales diferencias con las cookies es que estas envían el su contenido al servidor en cada petición; con estos mecanismos xxxStorage, no, de hecho, la información solo podrá ser accedida desde el lado del cliente por lo que es posible almacenar una gran cantidad sin afectar al rendimiento de la aplicación web. Además, con viene recordar que todo ello se hace usando JavaScript.

Algunos escenarios de implementación y uso son los siguientes:

- Almacenamiento de datos. Los datos son almacenados en el cliente y pueden ser pasados al servidor por intervalos de tiempo en lugar de en tiempo real. Además, con localStorage los datos estarán disponibles entre peticiones y sesiones del navegador.
- Utilización fuera de línea. Como consecuencia de que la relación entre los datos almacenados y el navegador no se pierden entre sesiones (para el localStorage).
- Mejora del rendimiento. Se pueden almacenar datos estáticos (por ejemplo, imágenes en codificación Base64) que no serán nuevamente obtenidos mediante peticiones al servidor.
- En el caso del sessionStorage no existe relación entre lo almacenado en diferentes pestañas o ventanas de un mismo navegador.
- Con el objeto sessionStorage existe la seguridad de que los datos serán borrados una vez termine la sesión de la ventana que los ha utilizado.

#### SessionStorage

La especificación de sessionStorage permiten que las aplicaciones web agreguen información a este objeto el cual está accesible durante toda la sesión. Este objeto se instancia por sesión y ventana, por lo que dos pestañas del navegador abiertas al mismo tiempo y para un mismo sitio web pueden tener información distinta. Al cerrar la sesión se pierde la información.

A continuación, se verá un ejemplo del uso del objeto `sessionStorage`. Supóngase que se desea reservar un vuelo y se quiere elegir si se llevan maletas extras:

```
<html lang="es">
  <body>
    <h1>Gestión de Maleta Extra</h1>
    <label> <input type="checkbox" id="cbMaletaExtra" /> Maleta extra </label>
    <button id="btnGuardar">Guardar</button>
    <button id="btnVer">Ver</button>
    <p id="resultado"></p>
    <script>
      const cbMaletaExtra = document.getElementById("cbMaletaExtra");
      const btnGuardar = document.getElementById("btnGuardar");
      const btnVer = document.getElementById("btnVer");
      const resultado = document.getElementById("resultado");

      // Se guarda el dato en sessionStorage
      btnGuardar.addEventListener("click", () => {
        let maletaExtra = cbMaletaExtra.checked;
        sessionStorage.setItem("maleta_extra", maletaExtra);
        resultado.textContent =
          "Se ha guardado correctamente si tiene o no maleta extra.";
      });

      // Ver la preferencia guardada en sessionStorage
      btnVer.addEventListener("click", () => {
        let maletaExtra = sessionStorage.getItem("maleta_extra");

        if (maletaExtra) {
          let mensaje = `Maleta extra: ${maletaExtra}`;
          resultado.textContent = mensaje;
        } else {
          resultado.textContent = "No hay nada guardado.";
        }
      });
    </script>
  </body>
</html>
```

En definitiva, el funcionamiento es prácticamente idéntico a **localStorage** en lo que respecta a los métodos (**getItem**, **setItem**, **removeItem**, etc.) cambiando únicamente la persistencia de la información.

### 6.7.3 REACTCONTEXT

React Context es un mecanismo que permite definir valores para que sean compartidos dentro de un contexto o, lo que es lo mismo, un conjunto de componentes. El valor a compartir puede ser desde un valor simple a un objeto e incluso una función. La operativa consiste en crear un

contexto y aplicarlo a un componente padre que agrupe a varios componentes hijos. A partir de ahí, todos los componentes que se definen en este contexto y sus descendientes podrán compartirlo, en definitiva, tendrán acceso a él. Un uso típico del contexto puede ser el de los datos del usuario autenticado distinguiendo si está en una sesión abierta o no. De esta forma se pueden modificar los menús, cabeceras y componentes de una aplicación.

A continuación, se pasará a mostrar un ejemplo introductorio sencillo. En primer lugar, hay que crear un fichero que defina el contexto, por ejemplo, **AppContext.js**:

```
import React from "react";
export const valoresDefecto = {titulo: 'Prueba de contexto 1.0', color: 'red'}
export const AppContext = React.createContext(valoresDefecto);
```

Ahora se crearán dos componentes, un encabezado y un pie de página, los cuales usarán el contexto recién creado. Primero el **Encabezado.js**:

```
import { useContext } from "react";
import { AppContext } from "../AppContext";
const Encabezado = () => {
  const contexto = useContext(AppContext);
  return (
    <header style = {{ color: contexto.color}}>
      <h1>Esto es un encabezado</h1>
      <h2>{contexto.titulo}</h2>
    </header>
  )
}
export default Encabezado;
```

Como puede verse, el acceso al contexto se hace mediante el hook **useContext** extrayendo en este caso el título y el color. Por otra parte, el **Pie.js** tendrá el siguiente código:

```
import { useContext } from "react";
import { AppContext } from "../AppContext";
const Pie = () => {
  const contexto = useContext(AppContext)
  return(
    <footer>
      <small>Este es el pie de página {contexto.titulo}</small>
    </footer>
  )
}
export default Pie;
```

La clave aquí es que ambos componentes comparten un mismo contexto sin conocer uno la existencia del otro.

Obviamente, también hay que tener en cuenta el contexto en la aplicación principal o componente padre de estos dos. Por tanto, App.js puede quedar así:

```
import './App.css';
import Encabezado from './components/Encabezado';
import Pie from './components/Pie';
import { AppContext, valoresDefecto } from './AppContext';
function App() {
  return (
    <div className="App">
      <AppContext.Provider value={valoresDefecto}>
        <Encabezado />
        <div>Esto simplemente es contenido.</div>
        <Pie />
      </AppContext.Provider>
    </div>
  );
}
export default App;
```

En este código se ve cómo el contexto actúa como envoltorio del resto de componentes. Además, es imprescindible pasarle valores por defecto al contexto independientemente de que se haya hecho o no en la creación del mismo.



## 6.8 DEPURACIÓN Y DOCUMENTACIÓN DEL CÓDIGO

Anteriormente ya se han presentado herramientas como el depurador de los distintos navegadores y en el caso de React, la extensión **React Developer Tools**. En el caso de React hay que tener en cuenta que uno de los posibles errores consiste en introducir bucles infinitos en el hook **useEffect**.

En cuanto a la documentación, en React se puede usar una biblioteca llamada **PropTypes** que permite validar las **props** que se pasan a un componente, pero solo en desarrollo, no en producción (para no afectar al rendimiento). Su objetivo es asegurar que los datos enviados a un componente sean del tipo esperado, lo que ayuda a evitar errores y mejora la legibilidad y mantenimiento del código. En caso de mandar una **prop** con un tipo incorrecto, mostrará un mensaje de advertencia en la consola.

Para poder usar **PropTypes**, hay que proceder a su instalación ya que en las últimas versiones no viene incluido en el núcleo de React. Por tanto, hay que lanzar el siguiente comando:

```
npm install prop-types
```

Supóngase el siguiente componente:

```
import PropTypes from "prop-types";

const Saludo = ({ nombre }) => <p>Hola, {nombre}</p>;

// Se definen las propTypes
Saludo.propTypes = {
  nombre: PropTypes.string.isRequired, // La prop nombre debe ser una cadena de texto
  y es obligatoria
};

export default Saludo;
```

Bien, en este caso, pueden darse las siguientes situaciones:

- Se proporciona la variable nombre como cadena. El sistema saluda al valor de la cadena y no muestra ningún error en consola.
- Se proporciona la variable nombre como numérica (p.e. <Saludo nombre={123} />). El sistema muestra el número como cadena, pero aparece un aviso en la consola.
- No se proporciona la variable nombre. El sistema no anexa nada al saludo (cadena vacía) y el sistema muestra un error por consola.

Todo ello gracias a que PropTypes pide que sea una cadena y que, además, sea requerida, es decir, no se puede entregar como vacía. Proptypes actualmente se considera obsoleto ya que ahora React soporta TypeScript y este es mucho más tipado, de ahí que no se incluya en el núcleo de React aunque todavía se pueda utilizar pero solo en desarrollo. En la siguiente URL se pueden ver las posibilidades de la biblioteca:

<https://www.freecodecamp.org/news/how-to-use-proptypes-in-react/>

Otra posibilidad es pensarse si se puede migrar o comenzar un nuevo proyecto usando TypeScript en lugar de JavaScript precisamente por la naturaleza fuertemente tipada de aquel. Un ejemplo de componente de saludo en TypeScript sería el siguiente:

```
import React from "react";
interface SaludoProps {
  nombre: string;
}
const SaludoTS: React.FC<SaludoProps> = ({ nombre }) => {
  return <p>Hola, {nombre}</p>;
};
export default SaludoTS;
```

Como puede verse, en este caso se declara un interfaz que debe cumplir el propio componente y se indica de qué tipo debe ser la propiedad (en este caso, una cadena).

También existe una herramienta llamada Storybook que permite documentar y desarrollar componentes de forma aislada. Entre sus ventajas está el hecho de que se pueden crear ejemplos visuales de los componentes y de sus comportamientos. También permite documentar casos de uso específicos. Se puede consultar en la siguiente URL junto con varios tutoriales: <https://storybook.js.org/>

## ÍNDICE DE FIGURAS

1. Fecha vs su correspondiente UTC.....	4
2. Ejecución de instrucción periódicamente .....	13
3. Tarea que dura más que el intervalo .....	14
4. Ejecución periódica de una función sin efectos colaterales.....	15
6. React router en package.json .....	21
7. Uso de parámetros en rutas .....	23
8. Ejemplo de re-renderizado.....	27
5. Ejemplo de interfaz Material-UI.....	32
9. Esquema de funcionamiento de un reducer .....	33

## BIBLIOGRAFÍA - WEBGRAFÍA

Vara Mesa, J.M. et al (2012). *Desarrollo Web en Entorno Cliente*. Editorial Ra-Ma.

Altadill Izura, P.X. (2023). *React Práctico*. Editorial Anaya

Barklund, M. (2024). *React in depth*. Editorial O'Reilly.

Maza, A. (2024) *React: Ampliando conceptos*.

<https://openwebinars.net/academia/portada/react-intermedio/> OpenWebinars