Índice

4. Eventos y *Hooks*. Formularios

4.1. Módulos en *JavaScript*

Es muy común tener todo el código en un sólo fichero *JavaScript*, pero esto se vuelve complejo de gestionar en cuanto el código crece. Es mucho más fácil de manejar, como ocurre en otros lenguajes de programación, si el código se divide en ficheros de modo que cada código está ubicada en un fichero separado.

Ya se han estudiado métodos para organizar el código como las funciones, pero es posible el uso de módulos para separar este código en ficheros autónomos. ¿Eso no era posible con clases y funciones? En realidad no de manera eficiente. Cuando se accede a una página o aplicación web, en la mayoría de ocasiones, se accede a un servidor desde donde se está descargando código al navegador para ejecutarlo. La forma más extendida de trabajar era incluir varias etiquetas **`<script>`** en la página **`HTML`**. De esta forma se pueden tener varios ficheros *JavaScript* separados cada uno para una finalidad concreta. Sin embargo, este sistema termina siendo muy poco modular, ofrecía algunas desventajas y resultaba lento ya que sobrecargaba al cliente con múltiples peticiones.

En *ECMAScript* se introducen los **Módulos ES** o **ESM**, que permite la importación y exportación de datos entre diferentes ficheros eliminando las múltiples llamadas y cargando el código de forma más óptima. Para trabajar con módulos hay que conocer las palabras clave:

- **`export`**, que exporta datos (variables, funciones, clases...) del fichero actual e
- **`import`** , que importa datos (variables, funciones, clases...) desde otro fichero **`.js`** al actual

4.1.1. Exportación de módulos

Un fichero *JavaScript* no tiene módulo de exportación si no se usa un **`export`**. Existen varias formas de exportar datos a través de esta palabra reservada:

- **`export { feo }`**, crea el módulo de exportación (si es la primera vez que se ejecuta el código) y añade el elemento **`feo`** ,
- **`export { fea }`**, añade el elemento **`fea`** al módulo de exportación,
- **`export { n1, n2, n3 }`**, añade los elementos **`n1`** , **`n2`** y **`n3`** al módulo de exportación,
- **`export * from './feos.js'`**, añade todos los elementos del módulo de **`feos.js`** al módulo de exportación.

Resulta más intuitivo exportar módulos al final del fichero. Un ejemplo:

```
const sumar = (x,y) => { // También se puede usar export function sumar.
 return x+y;
}
const restar = (x,y) => { // También se puede usar export function restar.
 return x-y
}
export { sumar };
export { restar };
export { sumar as add }; // Se añade el nombre add a la funcion sumar.
```

4.1.2. Importación de módulos

La palabra clave **`import`** permite cargar un módulo de exportación desde otro fichero para utilizar dichos elementos en el código. Existen varias formas de importar código:

- **`import { nombre } from './feos.js'`**, importa el elemento nombre de `feos.js`,
- **`import { n1, n2, n3 } from './feos.js'`**, importa los elementos indicados desde `feos.js`,
- **`import * as objetoFeo from './feos.js'`**, importa todos los elementos de `feos.js` en el objeto **`objetoFeo`**,
- **`import './feos.js'`**, no importa elementos sólo ejecuta el código de `feos.js`.

Es posible renombrar elementos con **`import`** utilizando **`as`** seguido del nuevo nombre, como ocurría en **`export`**.

Tras la importación será posible el uso de las funciones, objetos, clases o valores como si se hubieran declarado en el mismo fichero.

4.1.3. Convenciones a la hora de trabajar con módulos

- Hay que tener muy en cuenta que si se importan módulos y el fichero que los importa se está ejecutando en un navegador será necesario especificar **`type="module"`** en la importación del fichero:

  ```
  <script type="module" src="./prueba.js"></script>
  ```

  Pero cuidado, `prueba.js` no contiene los módulos exportados, sino que es el que realiza la importación.

- Por norma general, a los archivos *JavaScript* con módulos tienen extensión `.js`, aunque también se pueden encontrar con otras extensiones como `.es2015` o `.mjs` (menos extendidas).

- Se aconseja utilizar las rutas **UNIX** (empieza por `.`, `..` o `/`) en los `export` e `import` ya que son las que tienen mejor soporte tanto en navegadores como en *NodeJS*. También se pueden indicar rutas absolutas para cargar directamente desde el navegador.

En resumen, las exportaciones e importaciones en *JavaScript* son una herramienta indispensable para escribir código de una forma organizada y productiva.

4.2. Eventos

Los eventos permiten asociar código a ciertas acciones que ocurren en el navegador para que cuando se produzcan (evento) se ejecute una función (respuesta al evento). Entre ambos se encuentra el manejador del evento que decide, en base a una lógica muy simple, si la respuesta (función) se ejecuta o no: si un manejador devuelve **`true`** (o no devuelve nada), se realiza el evento asociado. Si el manejador devuelve **`false`**, se cancela el evento y no se ejecuta la función asociada.

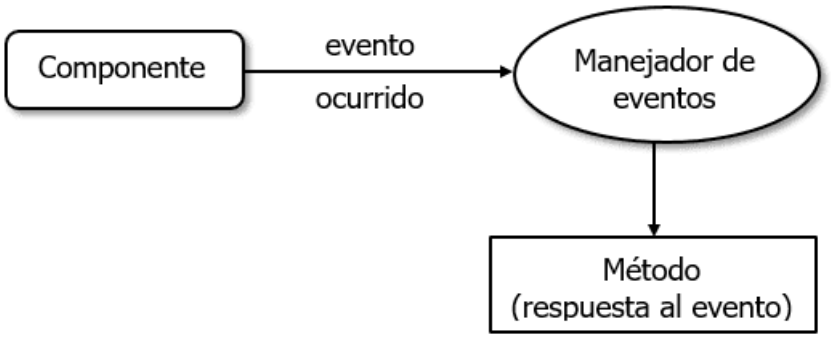

Figura 1: Diagrama del funcionamiento de un evento.

Los eventos más usados en *JavaScript* son:

- **`onFocus`**, al obtener un foco,
- **`onBlur`**, al salir del foco de un elemento,
- **`onChange`**, al hacer un cambio en un elemento,
- **`onClick`**, al hacer un clic en el elemento,
- **`onDblClick`**, al hacer doble clic en un elemento,
- **`onKeyDown`**, al pulsar una tecla (sin soltarla),
- **`onKeyUp`**, al soltar una tecla pulsada,
- **`onKeyPress`**, al pulsar una tecla,
- **`onLoad`**, al cargarse una página,
- **`onunLoad`**, al descargarse una página (salir de ella),
- **`onMouseDown`**, al hacer clic de ratón (sin soltarlo),
- **`onMouseUp`**, al soltar el botón del ratón previamente pulsado,
- **`onMouseOver`**, al entrar encima de un elemento con el ratón,
- **`onMouseOut`**, al salir de encima de un elemento con el ratón,
- **`onSubmit`**, al enviar los datos de un formulario,
- **`onReset`**, al reiniciar los datos de un formulario,
- **`onSelect`**, al seleccionar un texto,
- **`onResize`**, al modificar el tamaño de la página del navegador.

Más información sobre los eventos: aquí .

4.2.1. Asignación de eventos

La **primera forma**, y también la más sencilla, aunque **menos práctica** y **prohibida** a partir de ahora, de indicar que hay un evento asociado a un elemento **HTML** es indicándolo en la propia etiqueta del elemento:

```
<input type="button" value="Boton Hola Feo" onClick="alert('Hola Feo');alert('Adios Feo');"
/>
```

Otra opción es llamar a una función declaración (también **prohibida**):

```
<input type="button" value="Botón miFuncion" onClick="miFuncion('parametroFeo');" />
```

Cuando se ejecuta código dentro de un evento, se puede utilizar el objeto **this**. Este objeto es una referencia al elemento **DOM** donde se ha producido el evento, es decir, si el evento se ha producido al hacer clic a una imagen con **id="imagenFea"**, el objeto **this** referencia a esa imagen. Es equivalente a referenciar el objeto usando **document.getElementById("imagenFea")**:

```
<div id="cont" style="width:150px; height:60px; border:thin solid silver"
onMouseOver="document.getElementById('cont').style.borderColor='black';"
onMouseOut="document.getElementById('cont').style.borderColor='red';">
  Contenidos.
</div>
```

Equivalente con **this**:

```
<div id="cont" style="width:150px; height:60px; border:thin solid silver"
onmouseover="this.style.borderColor='black';" onmouseout="this.style.borderColor='red';">
  Contenidos
</div>
```

La **segunda forma** de asignación, más **eficiente,** es modificar mediante código el manejador de un evento predefinido en un documento **HTML**:

```
var elemento = document.getElementById("feo");  //Selección del elemento por su id.
 elemento.onClick = () => {
    this.style.backgroundColor = "red";
}; // Se asigna una función anónima.
```

También es posible asignar una **función declaración** o una **función flecha** del mismo modo. La ventaja de este método es evidente: es posible seleccionar un conjunto de elementos y asignarles métodos de forma simultánea. Ante el mismo evento, un objeto puede tener **sólo un manejador**, pero **varios *listeners*** (objetos que esperan la ejecución del evento). Hay que **tener cuidado** con la asignación de una función declaración:

```
function Saludar (x){
 console.log(`¡¡¡Hola ¡${x}!`);
};
var elemento = document.getElementById("feo");
elemento.onClick = "Saludar"; // Asigna la función Saludar como respuesta.
elemento.onClick = "Saludar()"; // Asigna el resultado de la función Saludar.
```

La **tercera forma** de hacerlo es a través del método **addEvenListener(evento, función, flujoEvento, opciones)**. El parámetro **flujoEvento** puede tomar los valores **true** para aplicar el modelo de **captura de eventos** o **false** para ***event bubbling*** (recomendado). El parámetro

opciones es un objeto que permite añadir particularidades a ese evento. Una interesante es la opción **once** que limita la ejecución del *callback* a una sola vez.

```
var cambioClase = function() {
 var clase = this.innerHTML.toLowerCase();
 document.body.className = clase; // Cambia la clase CSS por el nombre del botón.
}
var botones = document.getElementsByTagName("button"); // Todos los botones.
for (var i=0, len=botones.length; i<len; i=i+1) {
 botones[i].addEventListener("click", botonClick, false); // A cada botón.
}
```

Si se añade la opción **once** al evento, sólo se ejecutará una vez:

```
botones[i].addEventListener("click", botonClick, false, {once:true});
```

Para borrar un evento se usa su método contrario **removeEventListener(evento, función, flujoEvento).** Cabe destacar que la función a eliminar debe ser la misma que la utilizada al añadir el evento, por lo que no podemos usar funciones anónimas.

```
botones[i].removeEventListener("click", cambioClase, false);
```

Un evento especial: arrastrar y soltar (*drag and drop*)

A diferencia de los eventos típicos que funcionan de forma autónoma, para conseguir este efecto será necesario hacer referencia a varios eventos, siendo muy útil para el desarrollo de aplicaciones web. Más información sobre esta técnica aquí .

```
<!DOCTYPE HTML>
 <html>
    <head>
    <script>
       function allowDrop(evento) {
             evento.preventDefault();
       }
       function drag(evento) {
          evento.dataTransfer.setData("text", ev.target.id);
       }
       function drop(evento) {
          evento.preventDefault();
          var data = evento.dataTransfer.getData("text");
          evento.target.appendChild(document.getElementById(data));
       }
    </script>
 </head>
 <body>
    <div id="div1" ondrop="drop(event)" ondragover="allowDrop(event)"></div>
    <img id="drag1" src="img_logo.gif" draggable="true" ondragstart="drag(event)" width="336"
height="69">
 </body>
</html>
```

4.2.2. Flujo de eventos

Hay que tener en cuenta que un elemento del **DOM** que genera un evento puede estar dentro de otro que, a su vez, genera otro evento. A la hora de propagarse existen dos posibilidades al definir su flujo que permite saber cuál es el elemento que va a responder:

- con la **captura de eventos**, al pulsar sobre un elemento se produce una evento de arriba a abajo, desde el elemento **window**, pasando por **<body>** hasta llegar al elemento que lo captura
- en cambio, mediante el **burbujeo de eventos** (*event bubbling*) el evento se produce en el elemento de más abajo y va subiendo hasta llegar al **window**.

Según el siguiente fragmento **HTML**:

```
<html onclick="procesaEvento()">
 <head>
    <title>Ejemplo de flujo de eventos.</title>
 </head>
 <body onclick="procesaEvento()">
    <div onclick="procesaEvento()">Pincha aquí.</div>
 </body>
</html>
```

Cuando se pulsa sobre el texto **Pincha aquí** que se encuentra dentro del ** **, si el navegador sigue el modelo de *event bubbling*, se ejecutan los eventos en el orden que muestra el esquema:

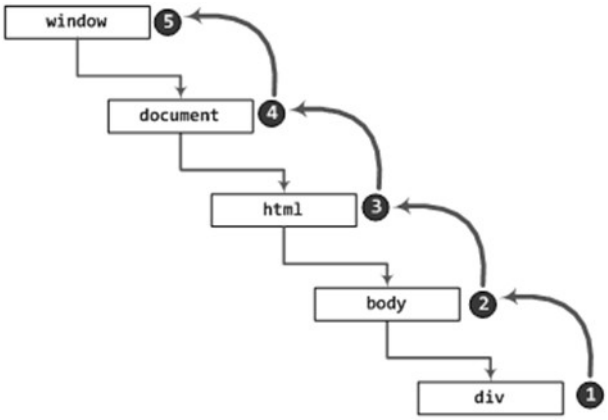

Figura 2: Flujo de eventos con Burbuja de eventos

En cambio mediante la captura de eventos sería:

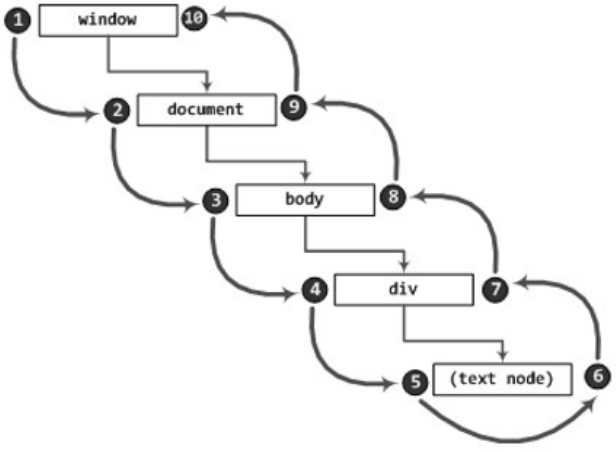

Figura 3: Flujo de eventos con Captura de eventos

El flujo de eventos definido en la especificación **DOM** soporta tanto el burbujeo como la captura, pero la captura de eventos se ejecuta en primer lugar. Los dos flujos de eventos recorren todos los objetos **DOM** desde el objeto **document** hasta el elemento más específico o viceversa.

4.2.3. Delegación de eventos

Este flujo a la hora de ejecutar eventos permite algo muy interesante que se puede utilizar para optimizar el código, se trata de la **delegación de eventos**. En el caso de tener un contenedor con varios elementos dentro a los que hay que colocar un evento, la idea inicial sería añadir un evento a cada uno de los elementos contenidos recorriendo uno a uno. La solución funciona de manera óptima con escasos elementos, pero ¿y si fueran un número elevado? El rendimiento caería hasta límites insoportables.

Además, si fuese necesario incluir nuevos elementos dentro del contenedor habría que tener en cuenta añadir el evento antes o después de introducirlo en el contenedor. Otra tarea extra.

Con la delegación de eventos (más concretamente con el flujo) si se añade un evento a un objeto éste podrá ser disparado por un elemento contenido, es decir, un elemento *child*. De esa forma tan sólo hay que añadir un evento al contenedor y no a cada uno de los elementos que contiene. Eso sí, si conviven diferentes tipos de elementos en el contenedor habrá que comprobar cuál ha sido el que ha generado el evento y actuar en consecuencia.

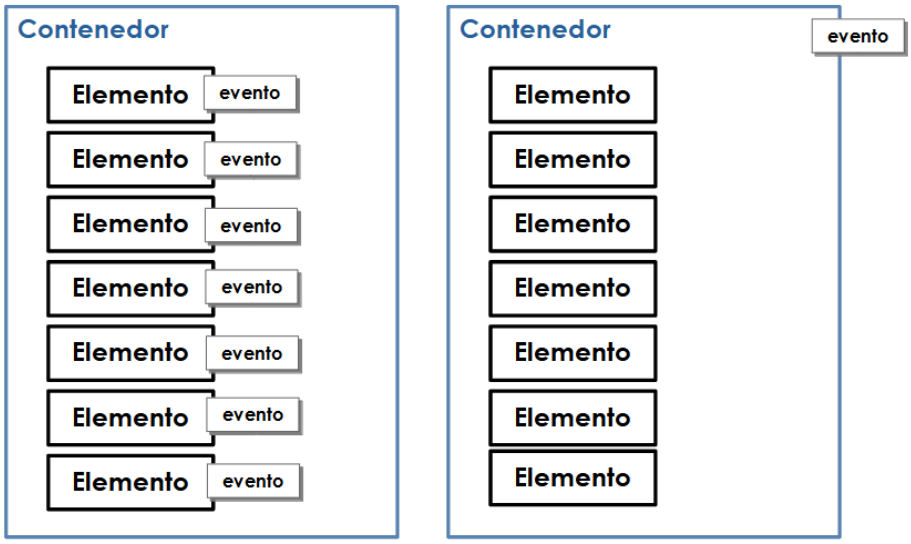

Figura 4: Ejemplo de delegación de eventos.

4.2.4. Esperar a la carga

Es buena idea comenzar a ejecutar código cuando se haya acabado de cargar una página **HTML**. Si se escribe código *JavaScript* que se ejecute al iniciar la página pero no se espera a que la página está completamente cargada, es posible obtener errores (a menudo aleatorios) del estilo de "**`no encuentro el elemento X del DOM`**" ya que el código *JavaScript* se ha ejecutado antes de la carga de dicho elemento.

Para evitar esta situación se utiliza el evento **`onLoad`** del objeto **`document`**. Cuando se produce este evento todos los elementos que forman la página web ya han sido cargados:

```
document.addEventListener('DOMContentLoaded', () => {
 // todo el código JavaScript aquí.
});
```

De este modo habrá que esperar al evento **`DOMContentLoaded`** para ejecutar el código que contiene, y este evento sólo ocurrirá cuando todos los objetos del **`DOM`** estén cargados.

También es posible esperar al evento **`onload`** del objeto **`window`**, el cual indica que todos los objetos de ventana (no sólo el **`document`**) han sido cargados:

```
window.onload = () => {
 // todo el código JavaScript aquí.
};
```

4.2.5. El objeto `event`

Cuando se crea una función como respuesta al producirse el evento el navegador automáticamente manda como parámetro un objeto de tipo **`event`**.

```
mostrarMensaje = (evento) => {
 console-log(evento.type);
}
```

Este objeto posee cierta información útil del evento que se ha producido. Sus atributos más destacados son:

- **`type`**, tipo de evento que es. Devuelve el nombre del evento tal cual, sin el **`on`**,
- **`keyCode`**, almacena el código de tecla de la tecla afectada por el evento,
- **`clientX`**/**`clientY`**, devuelve las coordenadas **`X`** e **`Y`** donde se encontraba el ratón, tomando como referencia al navegador,
- **`screenX`**/**`screenY`**, devuelve las coordenadas **`X`** e **`Y`** donde se encontraba el ratón, tomando como referencia la pantalla del ordenador.

```
function mostrarMensaje(evento){
 if(evento.type==="keyup"){
    console.log(evento.keyCode);
 } else if(evento.type==="click") {
    console.log(evento.clientX+" "+evento.clientY);
 }
}
document.getElementById("miObjeto").onclick=mostrarMensaje;
document.onkeyup=mostrarMensaje;
```

4.3. Formularios

Para poder interactuar con un formulario lo mejor es crearle un **`id`** y acceder a él con **`getElementById`**. Si el formulario no tiene atributo **`id`** pero sí **`name`**, se puede utilizar:

```
document.forms.nombreDelFormulario
```

Del mismo modo, para acceder a un campo del formulario bien se hace a través de su **`id`** o mediante la propiedad:

```
document.forms.nombreDelFormulario.nombreDelCampo
```

Teniendo el siguiente formulario:

```
<form name="formCliente" id="frmClnt">
 <fieldset id="infoPersonal">
 <legend>Datos Personales</legend>
    <p>
       <label for="nombre">Nombre</label>
       <input type="text" name="nombre" id="nom" />
    </p>
    <p>
       <label for="email">Email</label>
       <input type="email" name="correo" id="email" />
    </p>
 </fieldset>
</form>
```

Para acceder al formulario y al nombre del cliente:

```
var formulario = document.forms.formCliente; //Mediante el atributo name.
var correo = formulario.correo;
var formuId = document.getElementById("frmClnt");  //Mediante el atributo id.
var correoId = document.getElementById("email");
```

4.3.1. Elementos del formulario

No se tratarán todos los atributos de estas etiquetas ya que son demasiados y excede a los objetivos de este manual (se pueden consultar aquí). No obstante, se nombrarán los atributos necesarios para el correcto seguimiento de la unidad.

Texto

Para acceder al valor de un **input** de tipo texto simplemente hay que referenciar el atributo **value**.

```
<input type="text" id="miTexto">
<script>
 let elemento=document.getElementById("miTexto");
 console.log(elemento.value);
</script>
```

Existen variaciones de este elemento si se modifica el valor del campo **type** del siguiente modo:

- texto para búsquedas **<input type="search">**
- número de teléfono o móvil **<input type="tel">**
- dirección URL **<input type="url">**
- dirección de correo electrónico **<input type="email">**
- contraseña **<input type="password">**
- campo oculto (no mostrar al usuario) **<input type="hidden">**

No son todos los que **HTML5** ofrece, esos son consultables desde aquí.

Si es necesario escribir texto multilínea (o muy extenso) se utiliza la opción **<textarea>**. Este componente posee dos posee opciones para manejar su tamaño como **col** y **rows**.

```
<form id="formulario_feo">
 <textarea name="texto_feo" cols="100" rows="50" placeholder="Escribe aquí el texto...">
    Este es el texto por defecto de un componente textarea feo.
 </textarea>
</form>
```

Radio button

Los *radio button* son elementos del formulario que ante varias entradas permiten seleccionar sólo una de ellas. Se agrupan teniendo un **name** común. Se accede como un *array* donde se tiene el atributo **value** y el atributo **checked** que es **true** si está seleccionado o **false** en caso contrario.

```
<input type="radio" id="preguntaSI" name="pregunta" value="s&iacute;" />
<label for="preguntaSI">Sí</label>
<input type="radio" id="preguntaNO" name="pregunta" value="no" />
<label for="preguntaNO">NO</label>
<script>
 let elementos=document.getElementsByName("pregunta");
 for(let i=0;i<elementos.length;i++){
    if(elementos[i].checked === true)
    console.log(`Valor del elemento marcado ${elementos[i].value}.`);
 }
</script>
```

Checkbox

Similar a los *radio button* salvo que permite más de un elemento seleccionado. Para saber si un elemento *checkbox* ha sido seleccionado habrá que comprobar el estado de la propiedad **checked**:

```
<input type="checkbox" id="preguntaAS" name="pregunta" value="asc" />
<label for="preguntaAS">Piso con ascensor</label>
<input type="checkbox" id="preguntaAM" name="pregunta" value="amb" />
<label for="preguntaAM">Piso amueblado</label>
<script>
 let elementos=document.getElementsByName("pregunta");
 for(let i=0;i<elementos.length;i++){
    if(elementos[i].checked === true){
        console.log(`Valor del elemento marcado ${elementos[i].value}.`);
    }
 }
</script>
```

Select

Elemento que muestra un desplegable y permite elegir una opción. Destaca el atributo **options** que contiene un *array* con las opciones disponibles y el atributo **selectedIndex** que contiene la posición del *array options* seleccionada o **-1** si no hay seleccionada ninguna.

Dentro de cada **options**, **value** almacena el valor y **text** el texto mostrado.

```
<select id="aprobar" >
 <option value="10">Saco 10 en DWC.</option>
 <option value="9">Saco 9 en DWC.</option>
 <option value="8">Saco 8 en DWC.</option>
</select>
<script>


</script>
```

Evento `onSubmit`

Al enviarse un formulario se lanza el evento **onSubmit**. Si es necesario interrumpir el envío sólo hay que devolver **false** desde el manejador de eventos.

La estructura típica es la siguiente:

```
<form onSubmit="return validar();">
```

Si la función validar devuelve **true** se realiza el envío. Si devuelve **false** se cancela. Dentro de la función validar se hacen las validaciones que se estimen convenientes.

En algunas aplicaciones por motivos estéticos o de funcionalidad es deseable que al enviar un formulario no se haga desde un botón **submit**, sino desde cualquier otro evento que permita la ejecución de código. Hay que recordar que el evento **onSubmit** recarga la página y, en la mayoría de ocasiones, no es eso lo que se pretende.

Esto se puede hacer recogiendo el elemento del formulario y aplicando el método **`submit()`**.

```
<form id="formulario">
<script>
 let elemento=document.getElementById("formulario");
 elemento.submit();
</script>
```

4.3.2. Validación de formularios: expresiones regulares

Expresiones regulares con atributo `pattern`

Desde **HTML5**, los elementos de tipo **`input`** pueden definir un atributo llamado **`pattern`** donde definen una expresión regular.

```
<form action="/paginaDestino.php">
 <label for="pwd">Password:</label>
 <input type="password" id="pwd" name="pwd"
    pattern="(?=.*\d)(?=.*[a-z])(?=.*[A-Z]).{8,}"
    title="Debe contener al menos un número, una mayúscula y una minúscula. Además, debe
contener 8 o más caracteres.">
 <input type="submit">
</form>
```

En este ejemplo, si al enviar el formulario no se cumple el patrón definido, se mostrará el contenido del atributo **`title`**. Para ver más ejemplos: aquí .

Puedes ver un ejemplo de validación con **CSS** aquí . En la web **`http://html5pattern.com/`** hay una gran cantidad de expresiones regulares ya diseñadas para ser usadas con este atributo.

Funciones *JavaScript* para el uso de expresiones regulares

JavaScript posee dos formar de crear expresiones regulares, bien de forma literal con expresión regular, **`var re = /ab+c/`**, o bien con el objeto ***RegExp***, **`var re = new RegExp("ab+c")`**. Para comprobar si se cumple esa expresión regular se usa el método **`test`**. Este recibe una cadena y devuelve **`true`** si la cadena cumple esa expresión regular o **`false`** en caso contrario.

```
let re = new RegExp("ab+c");
let cadena=prompt("Dime una cadena.");
if(re.test(cadena)){
 alert("La cadena cumple el patrón de una a, entre 1 e infinitas b y al final una c.");
}
```

Sobre el uso de expresiones regulares hay más información aquí .

4.4. Ciclo de vida de un componente en `React`

Al igual que ocurre con los objetos en *JavaScript*, los componentes de **`React`** poseen un ciclo de vida basado en tres etapas:

- **montaje**, ocurre cuando se crea una instancia de un componente y se inserta en el **`DOM`**,
- **actualización**, sucede cuando se produce un cambio en las propiedades (**`props`**) o el estado (**`state`**) del componente,
- **desmontaje**, cuando el componente se elimina del **`DOM`**.

Cada una de estas fases provoca un evento que se puede utilizar para especificar el comportamiento del componente. En un inicio estos eventos tan sólo estaban disponibles en los componente de clase a través de **`componenteDidMount`**, **`componentDidiUpdate`** y **`componentWillUnmount`**, pero con la creación de los **`hooks`** es posible reproducir estos eventos en componentes funcionales (como se verá en los sucesivo).

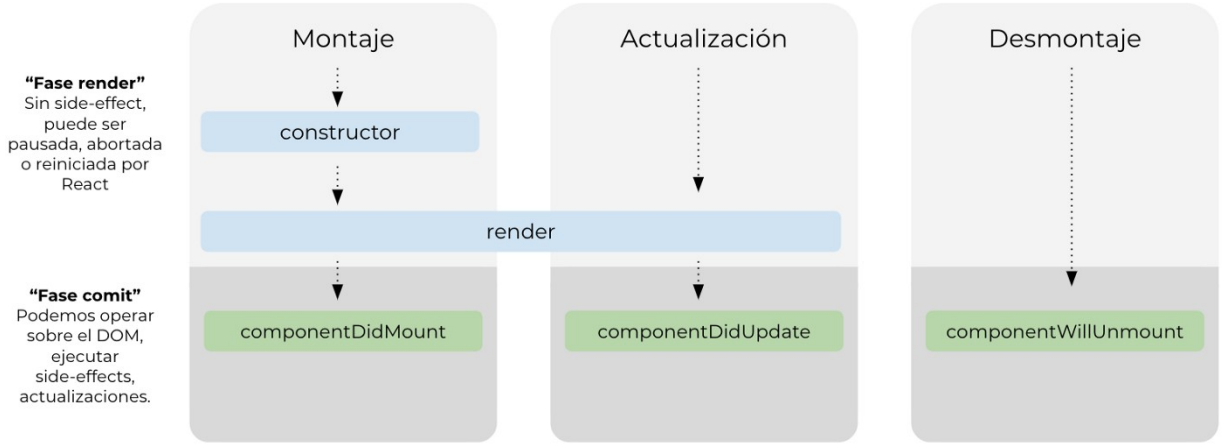

Figura 5: Estados del ciclo de vida de un componente.

4.4.1. Estado de un componente

Las propiedades de un componente no es un concepto ajeno a estas alturas ya que guarda muchas similitudes con los parámetros de una función, sin embargo, el concepto de **estado de la información** sí que puede ser nuevo. Se trata de la información que está entera y únicamente gestionada por el componente y que dicta su comportamiento, es decir, si se produce un cambio en esta información, el componente será dibujado (renderizado) de nuevo. Además, este estado lleva asociado una función (a modo de respuesta al evento) que es la encargada de modificarlo de manera exclusiva.

Pero, ¿cuándo usar **`props`** y cuándo **`state`**? Hay que tener en cuenta que un componente se comporta como una función determinista, es decir, pura. Una función con estas características no modifica los parámetros de entrada. Teniendo esto en mente:

- los componentes sin estado (sólo con **`props`**), no pueden cambiar su valor dentro del componente y son más fáciles de crear y mantener,
- los componentes con estado (**`props`** y **`state`**), permiten modificar el estado de forma interna y añadir el estado de la información al ciclo de vida del componente. Se utilizan para las comunicaciones cliente-servidor desde el exterior (desde un fichero, base de datos o API). Esto se debe a que el valor inicial de esta información es **`null`** o **`undefined`**

para contener, en algún momento del ciclo de vida, la información obtenida a través de un servicio externo a la aplicación.

Resumiendo:

- en cuanto a las propiedades de un objeto (**`props`**),
  - son los atributos que se envían desde el exterior del componente,
  - son de sólo lectura, por lo que nunca deben modificarse (funciones puras).
- en cuanto al estado de la información (**`state`**),
  - es un objeto de propiedades (**`clave = valor`**), un array o un valor primitivo, que es privada y que son controlados completamente por el componente,
  - se utilizan para obtener datos cliente-servidor ajenos al componente,
  - no se recomienda agregar un estado si no va a afectar al **`DOM`**, para el manejo de propiedades internas es mejor utilizar variables tradicionales,
  - todo estado lleva asociado una función que se utilizará para actualizar su valor, nunca se modificará directamente el valor del estado, esto no provocaría un redibujado del componente,
  - cuando se actualiza un estado, **`React`** vuelve a procesar el componente y sus elementos. Es importante saber que se puede retrasar la actualización del componente esperando a que otros puedan ser afectados, por lo que no se garantiza que los cambios de estado sean inmediatos.

4.5. Hooks en React

Uno de lo problemas que se plantean en la creación de componentes en **React** son las clases, ya que *JavaScript* no fue diseñado para soportarlas inicialmente. Estas suponen una dificultad no sólo para desarrolladores, sino para máquinas también. Por poner un ejemplo, es común la dificultad para entender la utilización del ámbito cuando se necesita referenciar un método con la palabra reservada **this**. En el caso de las máquinas también se identificaron algunos problemas para mejorar el rendimiento de los componentes debido a patrones que dificultan la optimización en el momento de la traducción.

Por lo tanto, existe la necesidad de un componente más simple que las clases capaz de tener estado y ciclo de vida. Para ello surgen los **hooks**, que son **funciones** que permiten "enganchar" el estado y el ciclo de vida de los componentes funcionales dotándolos de características propias de un componente de clase, es decir, proporcionan un estado y un ciclo de vida evitando el uso de las clases.

Los **hooks** **son funciones que te permiten "enganchar" el estado de React y el ciclo de vida desde componentes de función**. Los **hooks** no funcionan dentro de las clases, sino que te permiten usar **React** sin clases.

Más información sobre los **hooks** aquí .

4.5.1. El hook de estado: useState

Permite agregarle un estado local a un componente funcional.

```
import React, { useState } from 'react';
function MostrarTexto() {
 const [texto, setTexto] = useState(false);
 return (
   <div>
       <button type="button" onClick={() => setTexto(true)}>
        Mostrar Texto
       </button>
       <h1>
       { texto && ¡Se muestra el texto!}
       </h1>
 </div>
 );
}

export default MostrarTexto;
```

El componente **MostrarTexto** tiene un botón y una etiqueta **<h1>**, que depende del valor de la variable **texto** para ser visible. Hasta aquí nada nuevo.

Sin embargo, en la primera parte del componente, se ha utilizado el **useState**. En primer lugar se declaran dos variables utilizando *array destructuring*: la primera, **texto**, representa el estado del componente; la segunda, **setTexto**, representa la función con la que cambiaremos el valor de ese estado (lo que en un componente de clase se haría con **this.setState({...})**. Además, se le ha asignado un valor inicial (**false**).

Finalmente, se usa una respuesta para el evento **onClick** del botón que cambiar el estado al valor **true**, lo que permite que se muestre el texto.

4.5.2. El hook de efecto: `useEffect`.

Se trata de un **hook** que realiza varias acciones:

- reemplaza los métodos del ciclo de vida de los componentes de clase como **`ComponentDidMount`**, **`ComponentDidUpdate`** y **`ComponentWillUnmount`**,
- controla la ejecución de los "*side effects*" o efectos secundarios en programación (efectos que acompañan al principal).

`UseEffect` recibe dos parámetros:

- el primero **obligatorio**, una función que puede realizar cualquier tipo de operación,
- el segundo **opcional**, un **`array`** con las dependencias de ejecución.

Se ejecuta **siempre** después del primer dibujado y, en función de sus dependencias, después de cada actualización del componente.

El segundo parámetro se usa para especificar las dependencias. **`React`** comprobará si el valor de esas dependencias ha cambiado. Si es así, tiene que volver a ejecutar la función pasada como primer parámetro. En caso de no variar este valor, el **hook** no se ejecutará.

```
import React, { useState, useEffect } from "react";
function FormularioFeo() {
  const [nombre, setNombre] = useState("");
  const [apellidos, setApellidos] = useState("");

  useEffect(() => {
    console.log(`Se ha ejecutado el hook: ${nombre}.`);
  }, [nombre]);

  return (
    <div>
      <input value={nombre} onChange={(e) => setNombre(e.target.value)} />
      <input
        value={apelllidos}
        onChange={(e) => setApellidos(e.target.value)}
      />
    </div>
  );
}
```

En este caso se dispone de dos **`input`**: uno para establecer la variable **`nombre`** y el otro para **`apellidos`**. El **hook `useEffect`** simplemente realiza un **`console.log()`** en cada ocasión en que se ejecuta. Si no se le pasase el segundo parámetro **`[nombre]`**, se vería el resultado de **`console.log()`** cada vez que el usuario introduce un valor en cualquiera de los dos **`inputs`**. En cambio, al pasarle la variable **`nombre`** como dependencia, el **hook** se ejecuta solamente cuando el usuario introduce un valor en el **`input`** correspondiente a **`nombre`**.

Si lo que se necesita es que el **hook** se ejecute tan sólo una vez después del dibujado del componente, bastará con pasarle un **`array`** vacío. Si no se le pasa parámetro alguno, el hook se disparará en cada redibujado del componente.

Adeás, es posible utilizar varios **`useEffect`** en un mismo componente si se requieren diferentes acciones para varias dependendecias.

4.6. Formularios en React

El uso de formularios en desarrollo web es fundamental a la hora de crear aplicaciones interactivas ya que con ellos es posible validar y enviar información a los servidores. La verdad es que trabajar con formularios puede ser más desafiante de lo que parece ya que hay que tener muy en cuenta aspectos como la accesibilidad y la validación de los datos que se enviarán al servidor. Si bien es cierto que esta última no contribuirá a la seguridad e integridad de la información envida, de hecho, la validación de la información en formularios se limita a una cuestión de interfaz de usuario.

En **React** existen dos patrones de uso de formularios con usos muy diferenciados: formularios **controlados** y **no controlados**. No utilizarlos de forma adecuada para manipular la información puede dar como resultado la construcción de aplicaciones con problemas de escalabilidad, mantenimiento y/o rendimiento.

4.6.1. No controlados

Un componente no controlado es aquel que no usa el **state** o las **props** para representarse en el **DOM**, sino que usa la propia API del **DOM**. Se trabaja con los formularios de la misma manera que lo hace *Vanilla JavaScript*, es decir, seleccionando cada componente y accediendo a sus propiedades. Hay que recordar que en **React** está **prohibido el uso de esta API** (**getElementBy** y **querySelector**) para seleccionar elementos del **DOM**, por lo que habrá que utilizar el *hook* **useRef** como ya se ha estudiado con anterioridad.

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

const FormularioPruebaNC = () => {
 // Se crean las referencias a los objetos.
 const inputNombre = useRef(null);
 const inputCorreo = useRef(null);
 /*
  NOTA: Si el formulario es grande sería más efectivo hacer una referencia
  únicamente al formulario y obtener los datos a través de la función formData():
  const formData = new FormData(referenciaAlFormulario.current);
 */
 const manejarEnvio = (event) => {
  event.preventDefault();
  // Se obtienen los datos de los inputs no controlados (a través de sus referencias).
  const nombre = inputNombre.current.value;
  const correo = inputCorreo.current.value;
  // Se tratan los datos.
  console.log(`¡Hola ${nombre}!, tu correo electrónico es ${correo} .`);
  // Se limpian los inputs.
  inputNombre.current.value = "";
  inputCorreo.current.value = "";
 };

 return (
  <form>
   <div>
```

```
    <label htmlFor='nombre'>Nombre </label>
    <input type='text' id='nombre' ref={inputNombre} />
   </div>
   <div>
    <label htmlFor='correo'>Correo </label>
    <input type='email' id='correo' ref={inputCorreo} />
   </div>
   <button onClick={manejarEnvio}>Enviar</button>
  </form>
  );
};

export default FormularioPruebaNC;
```

Hay que tener en cuenta dos cosas: el componente que contiene el formulario **no se redibuja por sí mismo**, ya que no tiene cambios de estado y, además, **los valores de cada campo los gestiona el propio `DOM`** no **`React`**. Es una solución intermedia en la que intervienen dos API y para formularios pequeños no supone un coste alto en rendimiento.

4.6.2. Controlados

Un componente controlado es aquel que usa el **`state`** o las **`props`** para gestionar la información de los formularios. Más concretamente, es un componente que mantiene una sincronización entre el estado de **`React`** y el valor del campo. Si el estado cambia, el valor cambia (se produce un redibujado del componente).

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

const valoresIniciales = {
 correo: "",
 contrasena: "",
};

const FormularioPrueba = () => {
 const [datos, setDatos] = useState(valoresIniciales);

 function manejarEnvio(e) {
  // Se previene el comportamiento por defecto de lo formulario el cual recarga la página.
  e.preventDefault();
  // Aquí se usa "datos" para tratar la información.
  console.log(datos);
 }

 function manejarCambio(e) {
  // Deconstrucción del objeto "target" del evento pasado como parámetro.
  const { name, value } = e.target;
  /*
   Se utiliza spread operator para copiar el estado actual,
   y se reemplaza el valor del objeto al que afecta el input que ejecutó el evento.
```

```
  */
  setDatos({ ...datos, [name]: value });
 }

 return (
  <form>
   <label htmlFor='correo'>Correo electrónico</label>
   <input id='correo' name='correo' value={datos.correo} onChange={manejarCambio} />
   <label htmlFor='contrasena'>Contraseña</label>
   <input id='contrasena' name='contrasena' value={datos.contrasena} onChange={manejarCambio} />
   <button onClick={manejarEnvio}>Enviar</button>
  </form>
 );
};

export default FormularioPrueba;
```

De este modo será la API de **`React`** la que controle la información del formulario y, al cambiar su estado, se producirá un redibujado del componente como respuesta de la forma tradicional.