Centro Integrado de Formación Profesional

AVILÉS

Principado de Asturias

UNIDAD 7:
INTERACCIÓN CON EL USUARIO: EVENTOS Y FORMULARIOS

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	**16/02/2025**	**Aprobado**	**Primera versión**

ÍNDICE

UNIDAD 7: INTERACCIÓN CON EL USUARIO: EVENTOS Y FORMULARIOS

7.1 MODELO DE GESTIÓN DE EVENTOS. TIPOS. MANEJADORES DE EVENTOS

Una de las características básicas de JavaScript es la capacidad que tiene para capturar determinadas acciones producidas por el usuario, conocidas habitualmente como eventos. JavaScript detecta determinadas acciones producidas por el usuario, como son pulsar un objeto, enviar un formulario, modificar la información de un campo del formulario, movimiento del ratón sobre un objeto, etc. También detecta algunos eventos no relacionados directamente con acciones de usuario, como la carga o descarga de un documento en el navegador. Cada evento tiene asociado un nombre (onClick, onSubmit, onLoad, etc.), de forma que desde una página HTML se puede indicar qué acción se desea realizar cuando se genere un determinado evento en un elemento de la página. Por ejemplo, si se dispone de una página en la que se ha definido un formulario, es posible indicar que cuando se pulse el botón "Submit" se ejecute un script que valide los datos del formulario en la máquina del cliente y si todo es correcto que los envíe al servidor web para que este los procese.

Anteriormente ya se ha visto cómo manejar eventos y asociarlos a un elemento del DOM mediante addEventListener. La concepción clásica del manejo de eventos por HTML consiste en asociar el manejador directamente desde el componente mediante un atributo con el mismo nombre. En el caso del evento click, el manejador sería onClick. Ejemplo:

```
<img src="mundo.jpg" onclick="alert('Click en imagen');">
```

La especificación DOM define cuatro grupos de eventos dividiéndolos según su origen:

- Eventos del ratón: click, dblclick, mousedown, mouseup, mouseover, mousemove
- Eventos del teclado: keydown, keypress, keyup,
- Eventos HTML: load, unload, abort, error, select, change, submit, reset, resize, etc.

En la siguiente tabla se puede consultar una lista de los eventos que pueden darse en elementos HTML junto con su descripción.

Evento	Descripción
onload	Al terminar la carga de una página
onunload	Cuando se elimina un documento de la ventana
onabort	Cuando el usuario detiene la descarga de un elemento antes de que haya terminado
onerror	Cuando se ha producido un error en JavaScript
onreset	Cuando se pulsa un botón de reset
onclick	Cuando se hace clic sobre un elemento
ondblclick	Cuando se hace doble clic sobre un elemento
onmousedown	Cuando se pulsa con el ratón sobre un elemento
onmouseup	Cuando se levanta el botón del ratón de un elemento

onmouseover	Cuando el ratón entra dentro de un elemento
onmousemove	Cuando el ratón se mueve estando dentro de un elemento
onmouseout	Cuando el ratón sale de la zona de un elemento
onfocus	Cuando se activa el foco en un elemento
onblur	Cuando un elemento deja de estar activo
onkeypress	Cuando una tecla se pulse y levanta sobre un elemento
onkeydown	Cuando una tecla es pulsada sobre un elemento
onkeyup	Cuando una tecla es levantada de un elemento
onsubmit	Cuando se envía la información de un formulario
onreset	Cuando se limpia el contenido de un formulario
onselect	Cuando el usuario selecciona texto de una entrada
onchange	Cuando el elemento cambia su valor

7.1 GESTIÓN AVANZADA DE EVENTOS

A continuación, se va a implementar una API que permita gestionar los eventos típicos de una web (onclick, onmouseover, etc.) de forma moderna y avanzada. Primeramente, se define un objeto. Su nombre es **EventUtil** e inicialmente está vacío:

```
var EventUtil = { };
```

En el siguiente paso se crea el método **addEventHandler** que es el que los objetos emplearán para capturar los eventos producidos mientras un usuario lee o interacciona con una web.

```
EventUtil.addEventHandler = function (target, eventType, fnHandler)
{
    if (target.addEventListener)
        target.addEventListener(eventType, fnHandler, false);
    else if (target.attachEvent)
        target.attachEvent("on" + eventType, fnHandler);
    else
        target["on" +eventType] = fnHandler;
};
```

addEventHandler recibe tres parámetros: primero el objeto que origina o recibe el evento, es decir, el enlace, botón, etc. El siguiente es el nombre del tipo de evento: click, mouseover, etc. El último, la función responsable de procesar el evento. De esta forma se puede reemplazar:

```
<a href="#" id="btnlnk" onclick="DoClick()">Clic aquí</a>
```

por

```
EventUtil.addEventHandler(document.getElementById("btnlnk", "click", DoClick);
```

Esta codificación funciona para todos los navegadores, ya que si implementa **addEventListener** es el primero que usa, pero en caso de que no sea así, siempre puede probar con **attachHandler** y si tampoco, se establece la función de forma manual. El siguiente método pretende ejecutar métodos de objetos en lugar de funciones simples, en el momento en que se disparen los eventos.

Su código:

```
EventUtil.addObjEventListener = function (target, eventType, object, fnObjMethod)
{
    let objImpl = object;
    let fnObjMethodImpl = fnObjMethod;
    function EventListenerImpl()
    {
        let evt = EventUtil.getEvent();
        fnObjMethodImpl.call(objImpl, evt);
    };
    EventUtil.addEventHandler(target, eventType, EventListenerImpl);
}
```

addObjEventListener recibe cuatro parámetros. Los dos primeros son los mismos del anterior método, mientras que los dos últimos son el objeto y el método de éste a ejecutar cuando el evento se dispare. Se usaría así:

```
let btnLnk = document.getElementById("btnlnk");
EventUtil.addObjEventListener(btnLnk, "click", msgBox, msgBox.showMsg);
```

El listado utiliza el método **call** genérico de los objetos **Function**. Puede ser interesante extender el uso de **addObjEventListener** de forma que pudiera llamarse con un número indeterminado de parámetros y que estos se pasaran tal cual al método del objeto establecido como **listener** del evento. Sería algo así:

```
EventUtil.addObjEventListener2 = function (target, eventType, object, fnObjMethod)
{
    //…
}
```

Al comienzo de la ejecución del método no sólo se guardan en variables locales el método y el objeto, sino que en un array se almacenan todos los parámetros adicionales con los que se pudiera haber llamado a addObjEventListener2.

```
let objImpl = object;
let fnObjMethodImpl = fnObjMethod;
let additionalArgs = null;
if (arguments.length>4){
    let additionalArgsLength = arguments.length -4;
    additionalArgs = new Array(iAdditionalArgsLength);
    for(let i=0; i<iAdditionalArgsLength; i++){
        additionalArgs[i]=arguments[i+4];
    }
}
```

Los parámetros adicionales son accesibles a través de la propiedad arguments que tienen todas las funciones en JavaScript.

Otro ejemplo se puede ver en el listado siguiente. Se construye una cadena de caracteres con el código JavaScript a ejecutar, que en realidad será la expresión **fnObjMethodImpl.call(objImpl, evt)** que antes se ejecutaba directamente, con la salvedad de

que ahora como cadena tiene más parámetros. Al final se emplea la función **eval** que se encarga de ejecutar el código en JavaScript. Destacar que **eval** se ejecuta fuera del flujo normal del script mientras que las variables siguen existiendo con sus valores en el momento en que se llamó a **addObjEventListener2**. El código es el siguiente:

```
function EventListenerImpl()
{
    let evt = EventUtil.getEvent();
    let exprToEval = 'fnObjMethodImpl.call(objImpl, evt';
    if (additionalArgs!=null)
    {
        for(let i=0; i<additionalArgs.length; i++){
            exprToEval+=', additionalArgs[' + i +']';
        }
    }
    exprToEval+=');';
    eval(exprToEval);
};
```

Esto es muy útil para expresiones como ésta:

```
<a href=”#” id=”btnlnk” onclick="DoClick('txt',2)">Clic aquí</a>
```

Ahora se capturaría así:

```
let btnLnk = document.getElementById("btnlnk");
EventUtil.addObjEventHandler2(btnLnk,"click2,myObj, myObj.click, 'txt',2);
```

7.2 CLOSURES

Una clausura o closure se puede definir como una expresión que se evalúa en tiempo de ejecución. A continuación, se verá un ejemplo de uso.

Se define un objeto **MsgBox** mediante notación JSON que tiene un atributo **msg** con el mensaje a mostrar y dos métodos: **setMsg** y **showMsg** que establecen y muestran el mensaje respectivamente. El código es el siguiente:

```
let MsgBox = {
      msg : null,
      setMsg: function (msg)
      {
            this.msg = msg;
      },
      showMsg: function()
      {
            var self = this;
            function delayedShowMsg(){
                  alert(self.msg);
            }
            setTimeout(delayedShowMsg, 1000);
      }
};
```

El método **showMsg** va a tener una particularidad: el mensaje se va a mostrar con un cierto retraso de un segundo después de que se ejecute el método. **self** es una referencia al propio objeto. En el método **delayedShowMsg** lo que se desea es mostrar el mensaje de alerta, pero no se sabe a priori dónde está ese mensaje. Esta función se ejecuta fuera del flujo normal de ejecución del programa y no se conoce qué variables ve. Hay que preguntarse entonces si estar dentro del código de **showMsg** implica que ve las variables de dicho método y las propiedades del objeto que lo contiene.

Esta expresión es un **closure**: cuando se ejecuta **delayedShowMsg** la variable **self** contiene una referencia válida al objeto **MsgBox** y a partir de ahí resulta fácil llegar a la propiedad **msg**. Observar que **self** es visible a pesar de que **delayedShowMsg** se ejecuta fuera de la línea estándar de ejecución y que **self** guarda el valor que tenía justo antes de definirse la función.

En este tipo de expresiones, uno de los errores más frecuentes consiste en el uso de la palabra reservada "this". Por ejemplo, de esta forma:

```
function delayedShowMsg(){
    alert(this.msg);
}
```

Esto no funciona porque en el contexto de ejecución **this** no tiene sentido, ya que se hace referencia a sí misma y no al objeto. El truco es usar esa variable **self** para referenciar al objeto aprovechando las ventajas de los **closure** de JavaScript.

Para probarlo, no hay más que asignar el mensaje y ejecutar el método que lo muestra.

```
MsgBox.setMsg("Hola, mundo con closures");
MsgBox.showMsg();
```

7.3 GESTIÓN DE EVENTOS EN REACT

El manejo de eventos en React, como ya se ha visto con anterioridad, no es exactamente igual al de Vanilla JavaScript o jQuery. Ello es debido a que usa un sistema conocido como eventos sintéticos (Synthetic Events). Este sistema es en realidad una herramienta muy potente para manejar las interacciones del usuario en distintos navegadores de forma eficiente y consistente.

Los eventos sintéticos son un envoltorio de los eventos nativos del navegador que permite su uso sin tener en cuenta en cuál de ellos se utiliza. Esto es porque ofrecen una API consistente que asegura que los eventos funcionen exactamente del mismo modo en cualquier navegador. Un objeto de evento sintético replica de forma bastante exacta la estructura de los eventos nativos DOM de forma estandarizada.

Las características más importantes de este tipo de eventos son:

- Compatibilidad entre navegadores. Proporcionan una interfaz consistente entre todos los navegadores eliminando la necesidad de estar pendientes de las particularidades de cada navegador

- Agrupación de eventos. React optimiza el rendimiento mediante la agrupación de eventos, lo que significa que los objetos de eventos sintéticos se reutilizan para lograr una mayor eficiencia. Una vez que se maneja un evento, el objeto de evento se borra para su reutilización. Sin embargo, si necesita acceder al evento de forma asíncrona, se puede llamar a event.persist() para evitar que React lo borre.
- Propagación de eventos. Como los eventos nativos DOM, los sintéticos siguen el modelo de propagación de eventos incluyendo la fase de captura y de burbuja. Esto permite manejar la delegación de eventos y los manejadores tanto en el elemento objetivo como en sus padres.
- Prevención del comportamiento por defecto. Se puede prevenir el comportamiento por defecto del navegador usando el método **event.preventDefault()** el cual funciona de forma similar a como lo hace Vanilla JavaScript.

Ejemplos sencillos de uso:

1. Escribiendo manejadores de evento. En este ejemplo se puede ver un manejador sencillo que responde al clic de un botón.

```
const manejaClick  = (event) => {
    console.log('Botón pulsado:', event);
}
<button onClick={manejaClick}>Púlsame</button>
```

2. Manejando eventos de formulario. En este caso se maneja la entrada del usuario en el control.

```
const manejaCambioInput  = (event) => {
    console.log('Valor del input:', event.target.value);
}
<input type="text" onChange={manejaCambioInput} />
```

3. Previniendo el comportamiento por defecto. Al igual que en Vanilla JavaScript, se evita que se envíe el formulario ya que es su comportamiento por defecto.

```
const manejaSubmit  = (event) => {
    event.preventDefault();
    console.log('Formulario enviado');
}
<form onSubmit={manejaSubmit}>
    <button type="submit">Enviar</button>
</form>
```

7.2 EL FORMULARIO. LOS OBJETOS DEL FORMULARIO. PROPIEDADES Y MÉTODOS DE UN FORMULARIO

Un formulario web sirve para enviar, tratar y recuperar datos que son enviado y recibidos entre un cliente y un servidor web. Cada elemento del formulario almacena un tipo de dato o acciona una de sus funcionalidades. Los formularios disponen de una arquitectura y en este contexto están enmarcados en el lenguaje HTML. Los formularios se definen con etiquetas **<form>**. Para que sea funcional, la etiqueta necesita inicializar dos atributos:

- action. Contiene la URL donde se redirigen los datos del formulario.
- Method. Indica el método por el cual el formulario envía los datos. Puede ser POST o GET.

Un ejemplo de formulario sencillo:

```
<html>
    <head><title>Ejemplo de formulario</title></head>
    <body>
        <h3>Formulario</h3>
        <form action="www.Web.es/formulario.php"method="post">
        ...
        </form>
    </body>
</html>
```

Uno de los elementos más utilizados en un formulario es el etiquetado como input. Algunos de sus atributos son:

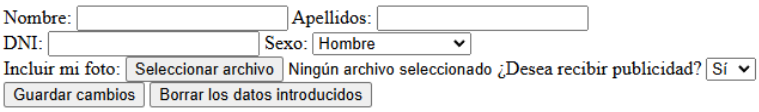
- type. Indica el tipo de elemento a definir. De él dependen el resto de parámetros. Los valores posibles que acepta el atributo type son:
    - text: cuadro de texto. Se puede usar un elemento textarea si se necesita más espacio
    - password: cuadro de contraseña, los caracteres aparecen ocultos tras asteriscos.
    - checkbox: casilla de verificación. Es mejor utilizar la etiqueta <select multiple>
    - radio: opción de entre dos o más. Su alternativa más moderna es <select>
    - submit: botón de envío del formulario. Mejor sustituirlo por <button type=”submit”>
    - reset: botón de vaciado de campos. Alternativa moderna: <button type=”reset”>
    - file: botón para buscar ficheros.
    - hidden: campo oculto para el usuario, ya que no lo visualiza en el formulario
    - image: botón de imagen en el formulario.
    - button: botón del formulario. La alternativa moderna es button ya que, además, ofrece más control sobre estilos y eventos.
- Name. El atributo name asigna un nombre al elemento. Si no se le asigna nombre a un elemento, el servidor no podrá identificarlo de la forma estándar y por tanto no podrá tener acceso al elemento, aunque sí en caso de usarse otras vías.
- Value. El atributo value inicializa el valor del elemento. Los valores dependerán del tipo de dato, en ocasiones los posibles valores a tomar serán verdadero o falso.

- Size. Este atributo asigna el tamaño inicial del elemento. El tamaño se indica en pixeles. En los campos text y password hace referencia al número de caracteres.
- Maxlength. Este atributo indica el número máximo de caracteres que pueden contener los elementos text y password. Es conveniente saber que el tamaño de los campos text y password es independiente del número de caracteres que acepta el campo.
- Checked. Este atributo es exclusivo de los elementos checkbox y radio. En él se define qué opción por defecto se quiere seleccionar.
- Disable. Este atributo hace que el elemento aparezca deshabilitado. En este caso el dato no se envía al servidor.
- Readonly. Este atributo sirve para bloquear el contenido del control, por tanto, el valor del elemento no se podrá modificar.
- Src. Este atributo es exclusivo para asignar una URL a una imagen que ha sido establecida como botón del formulario.
- Alt. El atributo alt incluye una pequeña descripción del elemento. Habitualmente y si no se ha desactivado cuando se posiciona el ratón encima del elemento (sin pulsar ningún botón), se puede visualizar la descripción de este.

Un ejemplo de formulario completo puede ser el siguiente:

```
<form action="pagina.php" method="post" enctype="multipart/form-data">
      <label for="nombre">Nombre:</label>
      <input type="text" id="nombre" name="nombre" maxlength="30" required />
      <label for="apellidos">Apellidos:</label>
      <input type="text" id="apellidos" name="ape" maxlength="80" required />
      <br />
      <label for="dni">DNI:</label>
      <input type="text" id="dni" name="dni" maxlength="9" pattern="\d{8}[A-Za-z]"
        required />
      <label for="sexo">Sexo:</label>
      <select id="sexo" name="sexo">
        <option value="hombre" selected>Hombre</option>
        <option value="mujer">Mujer</option>
        <option value="otro">Otro</option>
        <option value="no_especificar">Prefiero no decirlo</option>
      </select>
      <br />
      <label for="foto">Incluir mi foto:</label>
      <input type="file" id="foto" name="foto" accept="image/*" />
      <label for="publicidad">¿Desea recibir publicidad?</label>
      <select id="publicidad" name="publ">
        <option value="si" selected>Sí</option>
        <option value="no">No</option>
      </select>
      <br />
      <button type="submit">Guardar cambios</button>
      <button type="reset">Borrar los datos introducidos</button>
</form>
```

Formulario que tendría un aspecto similar al siguiente:



1. Ejemplo de formulario simple

FORMULARIOS EN REACT

Básicamente, la principal diferencia del uso de formularios en React respecto al JavaScript convencional o Vanilla JavaScript es que en este se manipulan directamente los valores de los campos mediante el DOM mientras que React se enfoca al estado (con useState como ya se ha visto) y en los manejadores de eventos. Aunque se pueden utilizar otros enfoques, el más utilizado y conveniente en React es el de formularios controlados. Este enfoque implica que los valores de inputs y otros elementos de interacción del usuario estén sincronizados con el estado o state del componente. Si se quiere modificar un campo del formulario, basta con modificar el estado y viceversa. En este código puede verse un ejemplo muy sencillo de formulario controlado mediante estados:

```
import React, { useState } from "react";
const FormControlado = () => {
  const [nombre, setNombre] = useState("");
  const [email, setEmail] = useState("");
  const cambioEntrada = (event) => {
    if (event.target.type === "text") setNombre(event.target.value);
    else if (event.target.type === "email") setEmail(event.target.value);
  };
  const envioForm = (event) => {
    event.preventDefault();
    alert(`Se ha enviado el formulario con el nombre ${nombre} y correo electrónico
${email}`);};
  return (
    <form onSubmit={envioForm}>
      <label>
        Nombre:
        <input type="text" value={nombre} onChange={cambioEntrada} />
      </label>
      <label>
        Correo electrónico:
        <input type="email" value={email} onChange={cambioEntrada} />
      </label>
      <button type="submit">Enviar datos</button>
    </form>
  );
};
export default FormControlado;
```

La alternativa menos recomendable a este enfoque es usar referencias (useRef) y acceder a los valores del virtual DOM. El problema es que en este caso no se aprovecha la naturaleza reactiva de la biblioteca.

7.3 UTILIZACIÓN DE FORMULARIOS DESDE CÓDIGO. ACCESO A LOS MIEMBROS DEL FORMULARIO

Cuando un campo está contenido en un elemento form, su elemento DOM tendrá una propiedad con el mismo nombre que se vincula al elemento DOM del formulario. Además, form dispone de una propiedad llamada elements que contiene una colección similar a una matriz con sus campos.

Como ya se mencionó, el atributo name de un campo de formulario determina la manera en que se identificará su valor cuando se envíe. También se puede usar como nombre de propiedad al acceder al elements. Un ejemplo:

```
<form action="ejemplo/submit.html">
      Nombre: <input type="text" name="nombre" /><br />
      Contraseña: <input type="password" name="pass" /><br />
      <button type="submit">Entrar</button>
</form>
<script>
      let formulario = document.querySelector('form');
      console.log(formulario.elements[1].type);
      console.log(formulario.elements.pass.type);
      // Ambas líneas de código muestran password
      console.log(formulario.elements.nombre.form == formulario);
      // Muestra true
</script>
```

Enviar un formulario implica que el navegador aparece a la página indicada en el atributo action, pero antes de que eso suceda, se activa un evento "submit" al que se le puede asociar un manejador que, además, evite que se lance el comportamiento predeterminado llamando a preventDefault en el objeto de evento.

```
<form action="ejemplo/submit.html">
      Valor: <input type="text" name="valor" /><br />
      <button type="submit">Guardar</button>
</form>
<script>
      let formulario = document.querySelector('form');
      formulario.addEventListener('submit', (e) => {
        console.log("Guardando valor", formulario.elements.valor.value);
        event.preventDefault();
      });
</script>
```

CAMPOS DE ARCHIVO

Este tipo de campo se diseñó inicialmente como una manera de cargar ficheros desde el equipo del usuario a través de un formulario. En los navegadores modernos también proporcionan una forma de leer dichos archivos desde programas JavaScript actuando como una especie de

guardián. El script no puede simplemente leer archivos privados de un ordenador, pero si se selecciona un archivo en dicho campo, el navegador interpreta que esa acción implica que el script puede leer dicho archivo. Véase un ejemplo de uso:

```
<input type="file" />
<script>
      let input = document.querySelector("input");
      input.addEventListener("change", () => {
        if (input.files.length > 0) {
          let fichero = input.files[0];
          console.log("Has elegido", fichero.name);
          if (fichero.type) console.log("Tiene como tipo ", fichero.type);
        }
      });
</script>
```

La propiedad files es un objeto similar a un array (sin serlo) que contiene los archivos elegidos en el campo. Inicialmente está vacío. La razón por la que existe es debida a que este tipo de componente puede admitir un atributo multiple lo que haría que se pudieran seleccionar varios archivos a la vez.

Los objetos de este files tienen propiedades como name (nombre de archivo), size (tamaño de archivo en bytes), type (tipo de archivo, p.e. text/plain o image/png). Lo que no tiene es una propiedad para el contenido del archivo. Llegar esto es algo más complicado. En primer lugar, hay que partir de que llevará un tiempo, por lo que la interfaz debe ser asíncrona para no congelar la página.

```
<input type="file" multiple />
<script>
      let input = document.querySelector("input");
      input.addEventListener("change", () => {
        Array.from(input.files).forEach((f) => {
          let reader = new FileReader();
          reader.addEventListener("load", () => {
            console.log(
              "Fichero",
              f.name,
              "comienza por",
              reader.result.slice(0, 20)
            );
          });
          reader.readAsText(f);
        });
      });
</script>
```

La lectura de un archivo se hace creando un objeto FileReader, registrando un manipulador de eventos “load” para él y llamando a su método readAsText para darle el archivo a leer. Una vez que acaba la carga, la propiedad result del lector alberga el contenido del archivo.

7.4 MODIFICACIÓN DE APARIENCIA Y COMPORTAMIENTO

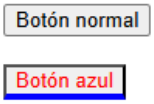
A través de hojas de estilo es posible mejorar notablemente el aspecto de los formularios. La visualización de un formulario puede depender de ciertas condiciones que vaya introduciendo el usuario. Por defecto los formularios tienen unos estilos asignados que tienen unos colores y unos bordes determinados. A continuación, se verán algunos ejemplos muy simples de aplicación de estilos a controles de un formulario.

En el siguiente ejemplo se modifica el color y borde de un botón:

```
<head>
    <style>
      .azul {
        color: red;
        border-bottom: 4px solid blue;
      }
    </style>
</head>
```

Para aplicar el estilo al botón se utiliza el procedimiento de asignación mediante una clase:

```
<button class="azul">Botón azul</button>
```



2. Comparación entre botón sin estilo aplicado y otro con texto rojo y borde inferior azul

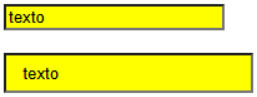
Si se quiere suavizar el aspecto de un cuadro de texto:

```
<style>
      .col {
        background-color: #ffff00;
      }
      .color {
        padding: 0.5em 1em;
        background-color: #ffff00;
      }
</style>
```

Se aplica cada uno de los estilos a un cuadro de texto:

```
<input class="col" type="text" value="texto" /><br /><br />
<input class="color" type="text" value="texto" />
```

En el primer componente simplemente se le aplica un color de fondo amarillo. Al segundo, se le añade un padding vertical de 0.5em y uno de 1em horizontal. Ambos componentes quedarían así:

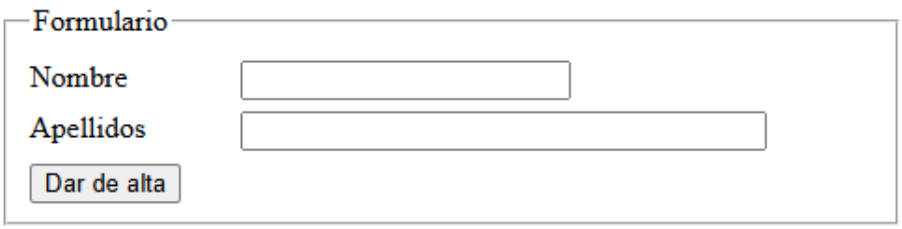


3. Cuadros de texto sin y con padding.

Otro ejemplo consiste en organizar los controles de un formulario mediante un elemento fieldset:

```
<html>
  <head>
    <style>
      div {
        margin: 0.4em 0;
      }
      div label {
        width: 25%;
        float: left;
      }
    </style>
  </head>
  <body>
    <fieldset>
      <legend>Formulario</legend>
      <div>
        <label for="nombre">Nombre</label>
        <input type="text" id="nombre" />
      </div>
      <div>
        <label for="apellidos">Apellidos</label>
        <input type="text" id="apellidos" size="35" />
      </div>
      <button class="btn" type="submit">Dar de alta</button>
    </fieldset>
  </body>
</html>
```

El formulario quedaría así:



4. Formulario básico organizado

Los formularios tienen unas acciones predeterminadas por defecto. Sin embargo, es posible darle otro comportamiento a uno de sus elementos. Por ejemplo, se podría querer que los datos se envíen a URLs diferentes en base a un dato que introduzca el usuario:

```
const enviar = (form) => {
      form.action = form.alta.checked ? "paginas/alta.html" : "paginas/baja.html";
      form.submit()
}
```

7.5 VALIDACIÓN Y ENVÍO. PROCESAMIENTO DE DATOS EN UN FORMULARIO.

Anteriormente se ha visto que es posible acceder a cualquier elemento HTML mediante selectores o jQuery, lo que podría permitir crear funciones de validación del contenido de cualquier control que permita que el usuario introduzca información. HTML desde su versión 5 introduce elementos que permiten hacer una validación automática en sus controles. Por ejemplo, si se accede a la referencia del control <input> en la sección de atributos, existen ejemplos como max, que limita el valor máximo de una entrada numérica. Otros posibles atributos son maxlength, que determina el máximo tamaño de la cadena a introducir. Además, en la sección HTML DOM reference, se pueden consultar todos los posibles tipos de input, siendo el más clásico el que permite introducir texto, pero como puede verse, se encuentran tipos personalizados para hora, fecha, color, etc. lo que permite limitar el tipo de datos introducido. Algunos ejemplos de uso de estos controles se pueden ver a continuación.

El siguiente código, muestra un formulario con un cuadro de texto con una longitud máxima de 15.

```
<form>
  <table>
    <tr>
      <td>Nombre: </td>
      <td>
         <input type="text" name="nombre" id="nombre" maxlength="15" required>
      </td>
    </tr>
  </table>
</form>
```

Ahora se puede añadir un control para recoger la edad y limitarla a que esté entre 18 y 110 años.

```
<tr>
  <td>Edad: </td>
  <td>
    <input type="number" name="edad" id="edad" min="18" max="110" required>
  </td>
</tr>
```

Todas estas validaciones se comprueban al ir a hacer el submit del formulario, pero en un momento dado puede interesar personalizar esta validación.

En un fichero validar.js se incluirá la lógica de validación; dentro de él, se incluye una función en la que se asocia el manejador de evento para el botón.

```
const inicializar = () => document.getElementById('enviar').addEventListener('click',
validar, false);
```

Ahora se crearán los distintos métodos de validación de cada campo.

Antes de ello, hay que incluir las siguientes líneas en el encabezado:

```
<script src="./validacion.js"></script>
<style>
        .error{
            border: solid 2px #FFFF00;
        }
</style>
```

El estilo es el que determinará cómo quedará un control cuando esté infringiendo una regla de validación. A continuación de la tabla, se incluirá la línea donde irá el mensaje de error y los botones de envío y de reinicio.

```
<p id="mensajeError"></p>
<p>
      <input type="submit" value="Enviar" id="enviar"></input>
      <input type="reset" value="Borrar" id="borrar"></input>
</p>
```

En el archivo JavaScript se crea un método flexible para la validación de un elemento dado por su nombre. La clave es la función **checkValidity** la cual comprueba si el control cumple con las reglas de validación proporcionadas:

```
const validaElemento = (nombre) => {
    let elemento = document.getElementById(nombre);
    if (!elemento.checkValidity()){
        error(elemento);
        return false;
    }
    return true;
};
```

El método error solo hace que ubicar el mensaje en el párrafo destinado a mostrarlo y le aplica la clase error. Se crea también otra función que es capaz de borrar el error y que no hace otra cosa que eliminar el estilo a los elementos del formulario.

```
const error = (elemento) => {
    document.getElementById('mensajeError').innerHTML = elemento.validationMessage;
    elemento.className ="error";
    elemento.focus();
};

const borrarError = () => {
    let formulario = document.forms[0];
    formulario.elements.forEach((e) => e.className="");
};
```

Finalmente, el método validar, el cual se llama antes del envío, comprueba todas las validaciones y, en caso de no cumplirlas, no hace el submit (e.preventDefault()).

```
const validar = (e) => {
    if (validaElemento("nombre") && validaElemento("edad") && confirm("Pulsa enviar
si quieres mandar el formulario"))
        return true;
    else{
        e.preventDefault();
        return false;
    }
};
```

Hasta aquí se ha visto cómo validar con los elementos que proporciona HTML5 y sus mensajes estándar, pero es bastante probable que se requiera modificar estos mensajes por unos más personalizados. Primeramente, se va a cambiar la función error para que pueda admitir un mensaje opcionalmente y, en ese caso, mostrarlo en el apartado correspondiente.

```
const error = (elemento, mensaje) => {
    document.getElementById('mensajeError').innerHTML = mensaje === "" ?
elemento.validationMessage : mensaje;
    elemento.className ="error";
    elemento.focus();
};
```

Se evalúa ahora la validez con la condición de "valueMissing" para darle mayor personalización. Se podría combinar esta validación con otras y precisamente este es el objetivo:

```
const validaElemento = (nombre) => {
    let elemento = document.getElementById(nombre);
    if (!elemento.checkValidity()){
        if (elemento.validity.valueMissing)
            error(elemento, "Debe introducir un nombre")
        return false;
    }
    return true;
};
```

Algunas de las bibliotecas más populares para la validación de formularios Javascript son:

- joi https://joi.dev/
- Validator.js https://github.com/validatorjs/validator.js
- Validate.js http://rickharrison.github.io/validate.js/
- jQuery Validation Plugin https://jqueryvalidation.org/documentation/

7.6 RUTINAS DEL LENGUAJE DE SCRIPT PARA LA APLICACIÓN A FORMULARIOS

Las rutinas de script son fragmentos de código reutilizables que permiten realizar tareas comunes relacionadas con formularios, como validación, procesamiento de datos, manipulación de campos o interacción con el usuario. Todas estas operaciones ya han sido vistas tanto en esta unidad como en anteriores. A continuación, se indican las operaciones más comunes realizadas con estas rutinas y los elementos de JavaScript para llevarlas a cabo:
1. Validación de campos de un formulario. En JavaScript este tipo de rutinas se basa en manipular directamente el DOM con métodos selectores como querySelector (o bibliotecas como jQuery) y addEventListener para la gestión de eventos. En React se pueden usar variables de estado o referencias directas a elementos del Virtual DOM (menos recomendable).
2. Procesamiento de datos. En este código se introduce la clase FormData

```
<html><head><script>
      const procesarDatos = (datos) => {
        return { nombreusuario: datos.nombre.trim().toUpperCase(),
          email: datos.email.toLowerCase()};
      };
      document.addEventListener("DOMContentLoaded", () => {
        const formulario = document.querySelector("#infoUsuario");
        formulario.addEventListener("submit", (e) => {
          e.preventDefault();
          const formData = new FormData(formulario);
          const datosProcesados = procesarDatos({
            nombre: formData.get("nombre"),
            email: formData.get("email")});
          console.log("Datos procesados:", datosProcesados);
          alert("Datos enviados correctamente.");});});</script></head>
  <body><form id="infoUsuario">
      <p><label for="nombre">Introduce tu nombre de usuario:</label>
        <input type="text" id="nombre" name="nombre" value="Juan Nadie" /></p>
      <p><label for="email">Introduce tu correo electrónico: </label>
        <input type="email" id="email" name="email" required /></p>
      <button type="submit">Enviar</button></form></body></html>
```

FormData es una clase nativa en JavaScript muy útil para obtener datos de un formulario. En este sentido, funciona como una especie de controlador de dicho elemento. Para ayudar a controlar el comportamiento y los datos de un formulario, la clase FormData en JavaScript recibe el nodo HTML, que contiene este componente form. Sobre este nodo, la clase permite ejecutar diversos métodos. En el ejemplo, se puede ver que se instancia un objeto de ese tipo proporcionando el formulario obtenido del DOM y mediante el método get pasando como parámetro el nombre del elemento de entrada, se obtiene su valor. ¿Se puede usar FormData en React? Aunque se pueda, no se recomienda ya que requiere un acceso al DOM y no es lo recomendable usando React. Lo que sí se puede es usar una variable de estado que en realidad sea un objeto al que se le puede llamar formData:

```
const [formData, setFormData] = useState({ nombre: "", email: "" });
```

A partir de este momento, se puede usar formData.nombre o formData.email en el atributo value:

```
<input type="text" name="nombre" value={formData.nombre} />
```

7.7 EXPRESIONES REGULARES.

Las expresiones regulares describen un conjunto de elementos que siguen un patrón; de hecho, es representan un lenguaje en sí mismas. Un ejemplo podría ser todas las palabras que comienzan por la letra "a" minúscula. JavaScript implementa el lenguaje de expresiones regulares y facilita las comprobaciones de ciertos datos que deben seguir una estructura concreta.

A continuación, se enumeran algunos de los caracteres especiales propios del lenguaje de expresiones regulares:
^ Principio de entrada o línea. Este carácter indica que las cadenas deberán comenzar por el siguiente carácter. Si este fuera una "a" minúscula, la expresión regular seria sería, ^a.
$ Fin de entrada o línea. Indica que la cadena debe terminar por el elemento precedido al dólar.
*** El carácter anterior 0 o más veces**. El asterisco indica que el carácter anterior se puede repetir en la cadena 0 o más veces.
+ El carácter anterior 1 o más veces. El símbolo más indica que el carácter anterior se puede repetir en la cadena una o más veces.
? El carácter anterior una vez como máximo. El símbolo interrogación indica que el carácter anterior se puede repetir en la cadena cero o una vez.
. Cualquier carácter individual. El símbolo punto indica que puede haber cualquier carácter individual salvo el de salto de línea.
x|y "x" ó "y": La barra vertical indica que puede ser el carácter "x" o el "y".
{n} n veces el carácter anterior. El carácter anterior a las llaves tiene que aparecer exactamente n veces.
{n,m} Entre n y m veces el carácter anterior. El carácter anterior a las llaves tiene que aparecer como mínimo n y como máximo m veces.
[abc] Cualquier carácter de los corchetes. En la cadena puede aparecer cualquier carácter que este incluido en los corchetes.
[^abc] Un carácter que no esté en los corchetes. En la cadena pueden aparecer todos los caracteres que no estén incluidos en los corchetes.
\b Fin de palabra. Este símbolo indica que tiene que haber un fin de palabra o retorno de carro.
\B No fin de palabra. El símbolo \B indica cualquiera que no sea un límite de palabra.
\d Cualquier carácter dígito. Este símbolo indica que puede haber cualquier carácter numérico, de 0 a 9.
\D Carácter que no es dígito. Este símbolo indica que puede haber cualquier carácter siempre que no sea numérico.
\f Salto de página. Este símbolo indica que tiene que haber un salto de página.
\n Salto de línea. Este símbolo indica que tiene que haber un salto de línea.
\r Retorno de carro. Este símbolo indica que tiene que haber un retorno de carro.
\s Cualquier espacio en blanco. Este símbolo indica que tiene que haber un carácter individual de espacio en blanco: espacios, tabulaciones, saltos de página o saltos de línea.

\S Carácter que no sea blanco. Este símbolo indica que tiene que haber cualquier carácter individual que no sea un espacio en blanco.
\t Tabulación. Este símbolo indica que tiene que haber cualquier tabulación.
\w Carácter alfanumérico. Este símbolo indica que puede haber cualquier carácter alfanumérico.
\W Carácter que no sea alfanumérico. Este símbolo indica que puede haber cualquier carácter que no sea alfanumérico.

Mediante este lenguaje, se pueden validar elementos de un formulario mediante expresiones regulares. Combinándolas, se pueden abordar infinidad de patrones de validación. Los más clásicos: correo electrónico, teléfono, código postal, NIF, etc.

Un ejemplo de método de validación de correo electrónico con expresiones regulares sería el siguiente:

```
const validaEmail = ()  => {
        let valor = document.getElementById("campo").value;
        return (/\w+([-+.']\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)/.test(valor));
}
```

Afortunadamente, como ya se vio anteriormente, HTML5 incorpora un tipo de input llamado email que hace ese trabajo sin necesidad de introducir un método que valide un correo electrónico:

```
<label for="email">Correo electrónico:</label>
<input type="email" id="email" name="email">
```

Otro ejemplo clásico es el del NIF. Para este caso tampoco es estrictamente necesaria la utilización de un método de validación, sino que se puede usar el atributo pattern del elemento input:

```
<label for="nif">NIF:</label>
<input type="text" id="nif" name="nif" pattern="^\d{8}[A-Za-z]$"
title="El DNI debe tener 8 números y 1 letra." />
```

Otro ejemplo: validación de un número de teléfono:

```
<label for="telefono">Número de teléfono:</label>
<input type="text" id="tel" name="tel" pattern="^\d{9}$"
title="El teléfono debe constar de 9 números exclusivamente." />>
```

Pero el uso de expresiones regulares no solo se limita a la validación de datos. Otra de las aplicaciones en las que se puede ver la potencia de este lenguaje es el filtrado de datos. Para ver cómo se aplica el filtrado, se va a desarrollar un ejemplo en React de filtrado de usuarios por nombre. Se creará un componente Filtrado (en Filtrado.js) con un formulario conteniendo un cuadro de texto de búsqueda por nombre. Los resultados aparecerán en una lista.

El código del componente es el siguiente:

```
import { useState } from "react";
const Filtrado = () => {
  const [busqueda, setBusqueda] = useState("");
  const usuarios = [
    { id: 1, nombre: "Cassandra Smith" },
    { id: 2, nombre: "Ryan Curtis" },
    { id: 3, nombre: "Dean Walker" },
  ];
  // Se crea una expresión regular desde la búsqueda
  const regex = new RegExp(busqueda, "i");
  // Se filtra la lista de usuarios
  const usuariosFiltrados = usuarios.filter((usuario) =>
    regex.test(usuario.nombre)
  );
  return (
    <>
      <input type="text" placeholder="Buscar usuario..." value={busqueda}
        onChange={(e) => setBusqueda(e.target.value)} />
      <ul>
        {usuariosFiltrados.map((usuario) => (
          <li key={usuario.id}>{usuario.nombre}</li>
        ))}
      </ul>
    </>
  );
};
export default Filtrado;
```

Aquí se usa la clase RegExp proporcionando como parámetros el patrón de búsqueda, el cual representa en sí mismo la expresión regular mientras que el segundo parámetro (“i”) indica que no tenga en cuenta mayúsculas ni minúsculas.

Por último, otra de las operaciones que puede realizarse utilizando expresiones regulares es el reemplazo de cadenas. Este tipo de datos en JavaScript tiene un método replace el cual se puede usar para sustituir parte de la cadena con otra. Por ejemplo:

```
console.log("papa".replace("p", "m")); // Se obtiene mapa porque solo reemplaza la primera ocurrencia
```

Lo importante es que el primer argumento puede ser una expresión regular. Además, si se quiere que se sustituyan todas las ocurrencias, no solo la primera, hay que añadir una opción g a la expresión regular:

```
console.log("Borobudur".replace(/[ou]/, "a")); // Barobudur
console.log("Borobudur".replace(/[ou]/g, "a")); // Barabadar
```

Sobre el uso de expresiones regulares se puede encontrar más información en:
https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Regular_expressions

7.8 PRUEBA Y DOCUMENTACIÓN DEL CÓDIGO

En anteriores unidades, se han tratado pruebas como las unitarias o las e2e. Continuando con esa línea, en esta unidad se verá cómo diseñar una prueba de integración para una aplicación React. Conviene recordar que son pruebas que aseguran que los componentes interactúan correctamente. Para el ejemplo, se utilizará un formulario similar a los ya vistos:

```
import { useState } from "react";
const Formulario = () => {
  const [nombre, setNombre] = useState("");
  const cambioEntrada = (event) => {
    setNombre(event.target.value);
  };
  const envioForm = (event) => {
    event.preventDefault();
    alert(`Se ha enviado el formulario con el nombre: ${nombre}`);
  };
  return (
    <form onSubmit={envioForm}>
      <label>
        Nombre:
        <input type="text" value={nombre} onChange={cambioEntrada} />
      </label>
      <button type="submit">Enviar datos</button>
    </form>
  );
};
export default Formulario;
```

Con anterioridad se vio que un archivo de pruebas debe tener el mismo nombre del componente anexionando el sufijo test, es decir, en este caso, Formulario.test.js

En la prueba, en primer lugar, se importan los objetos necesarios para realizarla, tanto del propio núcleo de reac tomo de la biblioteca de pruebas, así como el componente a probar. Al igual que con las unitarias, una prueba se define mediante un nombre y el código que va a ejecutar. Dentro de ella, se renderiza el componente y se obtienen los elementos de interacción del DOM, en este caso, el cuadro de texto y el botón. A continuación, se crea un mock que va a observar si se produce un alert en el componente. Un mock no es ni más ni menos que un objeto que simula un comportamiento de un sistema externo. Después se utiliza una función act para envolver aquellas interacciones que causan cambios de estado (en el ejemplo, en el nombre a introducir). Dentro de la función, se ejecuta un evento de usuario que escribe un texto y se dispara otro que simula el clic del botón.

Para saber si la prueba es correcta, se usa expect con el mock y un método **toHaveBeenCalledWith** que verifica si la salida del alert es la esperada. Al terminar la prueba, se lanza un restore sobre el mock.

El código de la prueba Formulario.test.js es el siguiente:

```
import { render, screen, fireEvent } from "@testing-library/react";
import { act } from "react";
import userEvent from "@testing-library/user-event";
import Formulario from "./Formulario";
test("El formulario se envía correctamente con el valor del input", () => {
  render(<Formulario />);// Se remderiza el componente Formulario
  // Se obtienen los elementos del DOM
  const txtNombre = screen.getByLabelText("Nombre:");
  const btnEnviar = screen.getByText("Enviar datos");
  // Se espía alert para poder capturar su llamada
  const mockAlert = jest.spyOn(window, "alert").mockImplementation(() => {});
  // Mediante act() se envuelven las interacciones que causan cambios de estado
  act(() => {
    userEvent.type(txtNombre, "Juan Pérez"); // Se escribe en el cuadro de texto
    fireEvent.click(btnEnviar); // Se simula el clic del botón
  });
  // Ahora se verifica que se haya llamado a alert con el valor correcto
  expect(mockAlert).toHaveBeenCalledWith(
    `Se ha enviado el formulario con el nombre: Juan Pérez`
  );
  // Se restaura la función alert original
  mockAlert.mockRestore();
});
```

Para probar si es correcta, al igual que en otro tipo de prueba, únicamente hay que lanzar **npm test** el cual debería dar una salida similar a la de la imagen:

```
 PASS  src/components/Formulario.test.js
  √ El formulario se envía correctamente con el valor del input (142 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        2.185 s
Ran all test suites.
```

5. Prueba de integración pasada correctamente

En cuanto a documentación, a continuación, se verá un ejemplo de uso de Storybook, una herramienta para crear documentación interactiva de componentes que ya se mencionó en anteriores unidades. Para usar Storybook, el primer paso es integrarla en el proyecto en el que se esté trabajando. Esto se hace mediante la siguiente orden:

```
npx storybook init
```

Básicamente, lo que hará una vez que se termine de instalar es crear un directorio **.storybook** y otro en **src/stories** en el que albergar las historias que se diseñen además de algunas que incluye a modo de ejemplo. Además, instalará las dependencias que necesite.

Antes de iniciar el servidor local en el que se mostrarán las distintas stories, se va a crear una personalizada con el ejemplo de formulario anterior, aunque hay que realizarle una serie de modificaciones.

La primera es cambiar su extensión a .jsx (Formulario.jsx) simplemente para que le sea más cómodo a Storybook trabajar con el componente. Además, hay que añadirle un par de props para que pueda ser compatible con las stories a diseñar. El código modificado sería el siguiente:

```
import { useState } from "react";
const Formulario = ({ initialState = "", onInputChange }) => {
  const [nombre, setNombre] = useState(initialState);
  const cambioEntrada = (event) => {
    const valor = event.target.value;
    setNombre(valor);
    if (onInputChange) {
      onInputChange(valor); // Notifica al padre si hay un manejador de cambios
    }
  };
  const envioForm = (event) => {
    event.preventDefault();
    alert(`Se ha enviado el formulario con el nombre: ${nombre}`);
  };
  return (
    <form onSubmit={envioForm}>
      <label>
        Nombre:
        <input type="text" value={nombre} onChange={cambioEntrada} />
      </label>
      <button type="submit">Enviar datos</button>
    </form>
  );
};
export default Formulario;
```

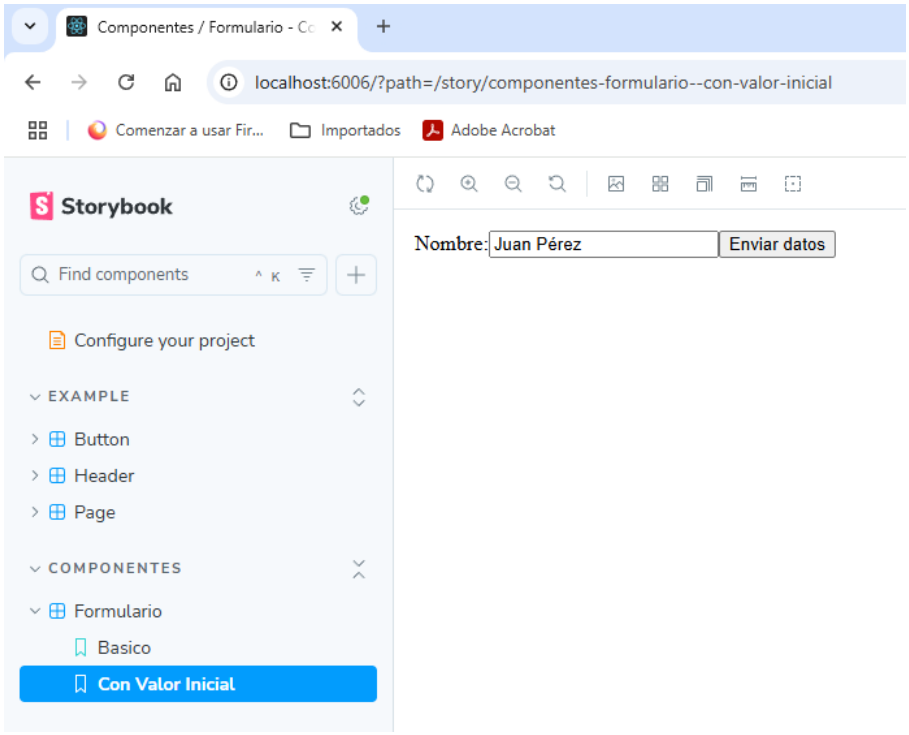
Ahora es cuando procede a crear la stories (como src/stories/Formulario.stories.js):

```
import React from "react";
import Formulario from "../components/Formulario";
export default {
  title: "Componentes/Formulario", // Título de la sección en Storybook
  component: Formulario, // El componente que se está documentando
};
// Historia básica
export const Basico = () => <Formulario />;
// Historia con valor inicial
export const ConValorInicial = () => <Formulario initialState="Juan Pérez" />;
```

Ahora se ejecuta al fin el storybook mediante la orden:

```
npm run storybook
```

En este momento se abrirá un servidor local con un interfaz web en el que pueden verse las distintas stories e incluso probar el componente de forma aislada:



6. Componente en Storybook

ÍNDICE DE FIGURAS

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

Haverbeke, M. (2024). *JavaScript elocuente*. 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

Patil, R. (2024) *How to Build Forms in React.* https://medium.com/@rashmipatil24/forms-in-react-d9029eeca5b5

Casero, A. (2024) *¿Qué es FormData en JavaScript?* https://keepcoding.io/blog/formdata-en-javascript/

Angular Minds (2024). *React Synthetic Events for Efficient Event Handling-* https://www.angularminds.com/blog/react-synthetic-events-for-efficient-event-handling