Centro Integrado de Formación Profesional

AVILÉS

Principado de Asturias

UNIDAD 8:
COMUNICACIÓN ASÍNCRONA

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	23/02/2025	Aprobado	Primera versión
1.1	06/03/2025	Aprobado	Corrección de errores
1.2	11/03/2025	Aprobado	Corrección de errores

ÍNDICE

UNIDAD 8: COMUNICACIÓN ASÍNCRONA

8.1 MECANISMOS DE COMUNICACIÓN ASÍNCRONA.

8.1.1 EL BUCLE DE EVENTOS

El entorno de ejecución de JavaScript ejecuta código en un solo hilo, lo que significa que solo puede ejecutar una parte de código a la vez. El código normalmente se coloca en la pila de llamadas antes de ejecutarse. Esta pila de llamadas es un segmento de memoria que registra el orden de las funciones en las que se llamaron.

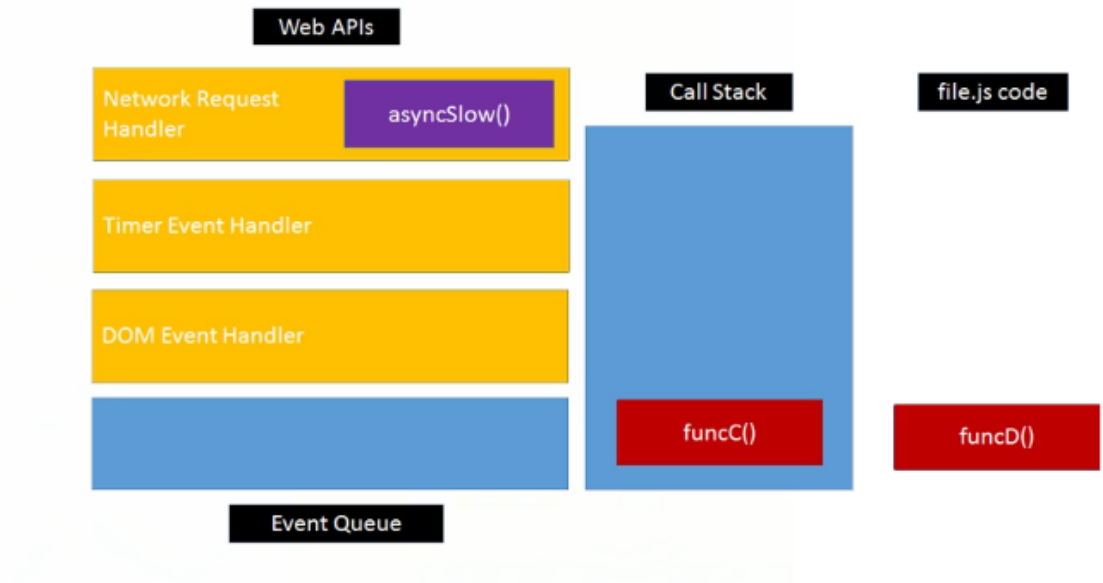
El bucle de eventos es el proceso en el que el navegador pone en cola las tareas y las ejecuta, una a la vez, colocándolas en la pila de llamadas. Ahora bien, ¿cómo hace esta operación el navegador? Al comienzo de una aplicación, todo el código JavaScript se ejecuta hasta su finalización y se coloca en la pila de llamadas en orden de ejecución. La cola de eventos está inicialmente vacía y, a medida que ocurren los eventos, los controladores de eventos colocan nuevas tareas en la cola de eventos. Algunos ejemplos de estos eventos son los clics del ratón, las pulsaciones del teclado y los eventos cronometrados.

Estas tareas esperan en la cola de eventos hasta que la pila de llamadas está vacía. Una vez que sucede, la primera tarea de la cola se coloca en la pila y las siguientes esperan hasta que la pila de llamadas esté vacía nuevamente, y el ciclo se repite. Esto es lo que se conoce como **bucle de eventos**.

¿Cómo encajan entonces las funciones asíncronas en el bucle de eventos? Cuando el entorno de ejecución de JavaScript encuentra una función asíncrona en la pila de llamadas, no la procesa inmediatamente; en lugar de bloquear la pila de llamadas hasta que finalice, permite que otro proceso maneje el procesamiento de esta función. Cuando el otro proceso ha finalizado, agrega una tarea nuevamente a la cola de eventos. Esta tarea es generalmente una función de devolución de llamada, que se pasa como uno de los argumentos a la función asincrónica original.

Anteriormente se ha dicho que el entorno de ejecución de JavaScript es de un solo subproceso, pero hay otros procesos que se ejecutan en un navegador, como temporizadores, controladores de entrada y API de solicitud de red que se ejecutan en paralelo con el bucle de eventos. Estos procesos paralelos se comunican con el citado bucle al colocar nuevas tareas en la cola de eventos.

Las funciones asíncronas son importantes porque algunas tareas, como las solicitudes de red, son lentas y los usuarios se molestarán si sus navegadores se bloquean debido a que una solicitud de red lenta impide que se ejecute otro código. En definitiva, permiten que las tareas lentas se ejecuten en un hilo separado para evitar que se bloquee el navegador. Las devoluciones de llamadas se utilizan para especificar qué nuevas tareas se envían de vuelta a la cola de eventos una vez que el otro proceso ha terminado de procesar la tarea original.

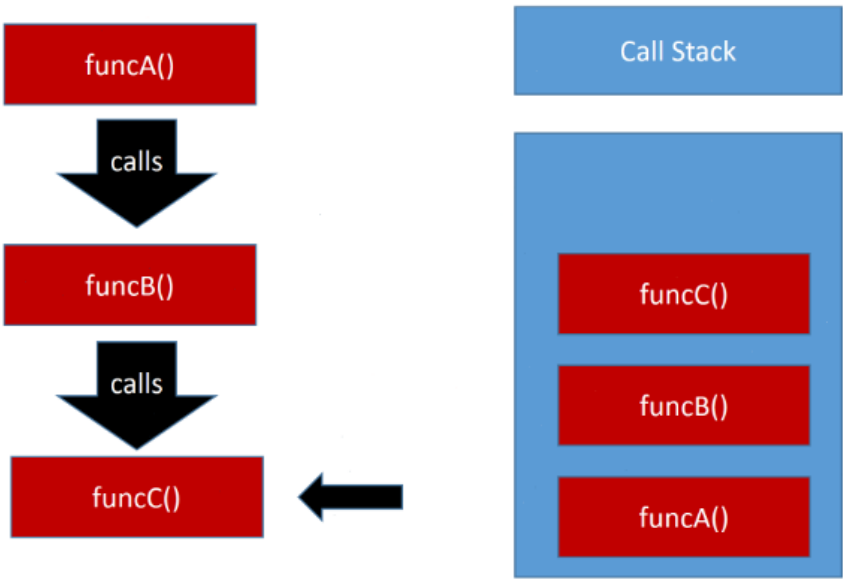


1. Procesado de función asíncrona

8.1.2 LA PILA DE EJECUCIÓN

La pila de ejecución (o de llamadas) rastrea las funciones que están actualmente activas y siendo procesadas. Funciona de la siguiente forma:

- Cuando se produce una llamada a una función, se coloca en la pila
- Cada una de las funciones adicionales dentro de la función original se ubican en la parte superior de la pila de ejecución.
- Cuando una función termina su ejecución, se saca de la pila y se procesa la siguiente función. Conviene darse cuenta de que la pila mantiene una pista del orden de las llamadas a las funciones.



2. Pila de llamadas

Véase el siguiente código:

```
function funcA() {
  funcB();
}
function funcB() {
  funcC();
}
function funcC() {
  console.log(Error().stack); //Error se usa solo para mostrar la pila de llamadas
}
funcA();
/* Salida en consola
 "Error
     at funcC (example.js:15:17) <-- funcC está en la parte de arriba de la pila
porque se llamó la última
     at funcB (example.js:12:5)
     at funcA (example.js:9:5)   <-- funcA está en la parte inferor ya que se llamó
primero
     at example.js:17:1"
 */
```

Stack Overflow (Sobrecarga de la Pila)

Si la pila crece demasiado y excede la cantidad de memoria reservada, se producirá un error de sobrecarga de la pila (stack overflow). Esto pasa frecuentemente cuando una función se llama a sí misma de forma recursiva. Es muy sencillo de simular, simplemente basta con crear una función recursiva sin condición de salida.

```
function funcA(){
    funcA();
 }
 funcA();
// Provoca un desbordamiento de pila ya que funcA() se llama recursivamente sin
condición de salida
```

8.1.3 LA COLA DE EVENTOS

La cola de eventos mantiene un seguimiento de las tareas que están esperando ser puestas en la pila de llamadas para ser ejecutadas. Se añaden por APIs Web que trabajan en paralelo con el entorno de ejecución de JavaScript.

Se pueden encontrar tres ejemplos de APIs Web que añaden tareas a la cola de eventos:

- Temporizadores. Programan tareas a ser añadidas a la cola de eventos
- Manejadores de eventos DOM. Las interacciones del usuario (clics del ratón o pulsaciones del teclado) se gestionan colocando tareas en la cola de eventos.
- Peticiones de red. Se procesan asíncronamente y devuelven resultados colocando tareas en la cola de eventos.

Todo ello lleva a que cuando la pila de llamadas se vacía, tome la primera tarea de la cola de eventos y se procese. Las tareas restantes en la cola esperan hasta que la pila de llamadas vuelva a vaciarse nuevamente, con lo que se tiene el ya citado bucle de eventos.

8.1.4 PROGRAMACIÓN ASÍNCRONA

La programación asíncrona en JavaScript se consigue utilizando APIs Web que procesen código en hilos separados. Estas APIs devuelven sus resultados procesados como tareas en la cola de eventos, definidas como funciones callback y pasadas dentro de las APIs Web, lo que permite a JavaScript conseguir multihilo en una ejecución de un solo hilo.

Para distinguir programación síncrona de asíncrona hay que imaginarse intentando ejecutar una tarea lenta de forma síncrona. Esto llevará un montón de tiempo en terminar su procesamiento e impedirá a otras tareas ejecutarse. En el código siguiente, nótese que la tareaLenta() lleva un tiempo largo de procesado impidiendo a otras más rápidas su ejecución:

```
function tareaLenta(){
    /* Tarda 2 segundos en ser procesada */
    let now = new Date().getTime();
    while(new Date().getTime() < now + 2000){ /* Procesando */ }
    console.log("Tarea lenta finalizada");
}
function tareaRapida(){
    console.log("Tarea rápida finalizada")
}

tareaRapida();
tareaLenta();
tareaRapida();
tareaRapida();
tareaRapida();

/*  Salida en consola:
    > "Tarea rápida finalizada"
    ....2 segundos después
    > "Tarea lenta finalizada"
    > "Tarea rápida finalizada"
    > "Tarea rápida finalizada"
    > "Tarea rápida finalizada"
*/
```

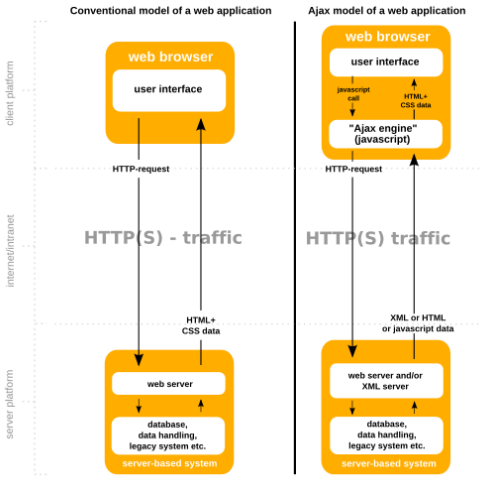
Precisamente, lo bueno de la programación asíncrona es que impide que las tareas lentas bloqueen a las rápidas en su procesado. El código asíncrono se ejecutará solo cuando la pila de llamadas esté vacía. En el código de prueba se simula la asincronía usando un temporizador, el cual no ejecuta el código en el hilo principal sino en uno secundario.

El siguiente ejemplo ilustra el procesamiento asíncrono:

```
function tareaLenta() {
  console.log("Tarea lenta finalizada");
}
function tareaLentaAsinc(val) {
  setTimeout(tareaLenta, 2000); // Termina en dos segundos pero en un hilo aparte
}
function tareaRapida() {
  console.log("Tarea rápida finalizada");
}
tareaRapida();
tareaLentaAsinc();
tareaRapida();
tareaLentaAsinc();
tareaRapida();
tareaRapida();
/*  Salida en consola:
    > "Tarea rápida finalizada" <--- Las tareas rápidas se procesan primero
    > "Tarea rápida finalizada"
    > "Tarea rápida finalizada"
    > "Tarea rápida finalizada"
    > "Tarea lenta finalizada" <--- Las tareas lenta separadamente y no bloquean la
pila de ejecución
    > "Tarea lenta finalizada"
*/
```

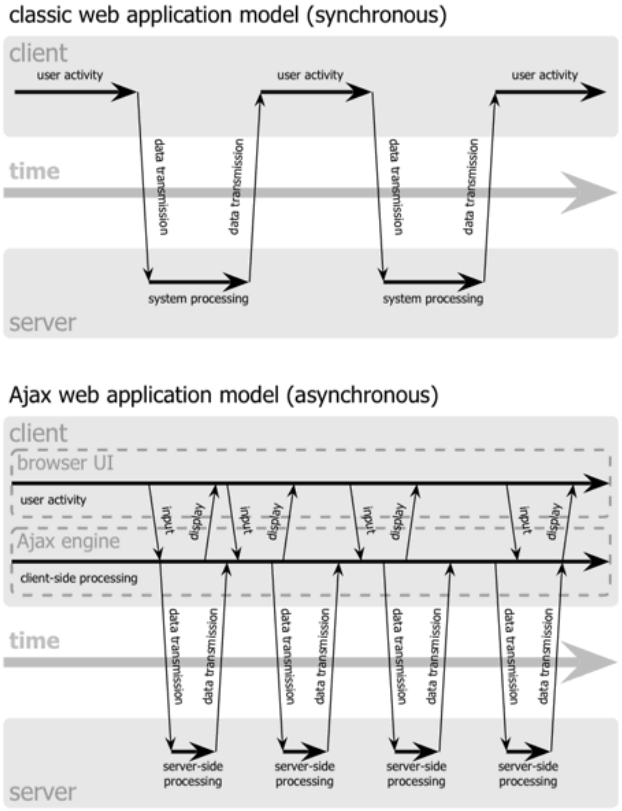
8.1.5 AJAX

AJAX es el acrónimo de Asynchronous Javascript And XML (Javascript asíncrono y XML) y es un mecanismo para hacer peticiones asíncronas al servidor desde Javascript. Básicamente Ajax permite poder mostrar nuevos datos enviados por el servidor sin tener que recargar la página, que continuará disponible mientras se reciben y procesan los datos enviados por el servidor en segundo plano.



3. Funcionamiento de Ajax

Sin Ajax cada vez que se necesiten nuevos datos del servidor la página deja de estar disponible para el usuario hasta que se recarga con lo que envía el servidor, tal como se vio en la comunicación síncrona. Con Ajax la página está siempre disponible para el usuario y simplemente se modifica (cambiando el DOM) cuando llegan los datos del servidor:



4. Comparación entre comunicación síncrona y asíncrona con Ajax

8.2 OBJETOS, PROPIEDADES Y MÉTODOS RELACIONADOS

8.2.1 EL OBJETO XMLHTTPREQUEST

El objeto principal sobre el que pivota toda comunicación Ajax es el llamado XmlHttpRequest. Para ilustrar el funcionamiento de este objeto, en el código siguiente se pide al usuario que haga clic en un botón, se toman datos del servidor usando técnicas Ajax y se muestra el texto en la misma página que el botón.

```
<html>
  <head>
    <title>Probando Ajax</title>
    <script>
      let XMLHttpRequestObject = new XMLHttpRequest();

      const getData = (dataSource, divID) => {
        event.preventDefault();
        if (XMLHttpRequestObject) {
          let obj = document.getElementById(divID);
          XMLHttpRequestObject.open("GET", dataSource);
          XMLHttpRequestObject.onreadystatechange = () => {
            if (XMLHttpRequestObject.readyState == 4)
              obj.innerHTML = XMLHttpRequestObject.responseText;
          };
          XMLHttpRequestObject.send(null);
        }
      };
    </script>
  </head>

  <body>
    <h1>Tomando datos con Ajax</h1>
    <form>
      <button onclick="getData('text.data','targetDiv')">
        Mostrar mensaje
      </button>
    </form>
    <div id="targetDiv">
      <p>Los datos devueltos irán aquí.</p>
    </div>
  </body>
</html>
```

Este ejemplo necesita un objeto XMLHttpRequest para comenzar, de tal forma que comienza con el código que creará el objeto; el cual se ejecuta inmediatamente cuando la página se carga. Al pulsar el botón se cargará el contenido del archivo text.data que debería estar en la misma ruta, pero también se puede probar a cargar un archivo externo.

En este getData, se comienza chequeando que realmente es un objeto válido en la variable XMLHttpRequestObject con una sentencia if. En este punto, se tiene un objeto XMLHttpRequest en la variable XMLHttpRequestObject.

Se puede configurar el objeto para usar una URL cualquiera con el método open. A continuación, se muestra la sintaxis del método open:

```
open('method', 'URL'[, asyncFlag[, 'userName'[, 'password']]])
```

Parámetro	**Significado**
Method	Se usa para abrir la conexión como GET, POST, PUT, HEAD o PROPFIND
URL	La URL solicitada
asyncFlag	Valor booleano que indica que la llamada es asíncrona. Por defecto es true
userName	Nombre de Usuario
password	Contraseña

La URL de la cual se quieren tratar los datos se le pasa a la función getData como el argumento dataSource. Para abrir una URL, se pueden usar técnicas estándar HTML como GET, POST o PUT. Cuando se usa Ajax, normalmente se utiliza principalmente GET para obtener datos, y POST para mandar una gran cantidad de datos al servidor, por lo cual en este ejemplo se utilizará GET para abrir el archivo text.data. Se configura el XMLHttpRequestObject para usar la URL especificada en este ejemplo, pero no se conecta todavía a este fichero. Por defecto, la conexión se hace asíncronamente, lo que significa que esta orden espera hasta que la conexión se haya hecho y los datos se hayan terminado de descargar. Se puede usar un tercer argumento opcional, asyncFlag en la llamada al método open para hacer la llamada síncrona, lo que implica que todo parará hasta que la llamada al método termine, pero las cosas no se hacen así en Ajax, evidentemente.

El objeto **XMLHttpRequest** tiene un evento **onreadystatechange** que permite manejar operaciones de carga asíncronas, así que basta con incluir el manejador de evento que se desee para poder trabajar de forma asíncrona. En el ejemplo se usa una función anónima la cual será llamada cuando el objeto **XMLHttpRequest** detecte algún cambio, como cuando se descargan datos. Aquí hay que examinar dos propiedades de este objeto: la propiedad **readyState** y la propiedad **status**. La primera informa de cómo van los datos que se están cargando. Los posibles valores que la propiedad **readyState** puede tomar son los siguientes:

- 0: No inicializado
- 1: Cargando
- 2: Cargado
- 3: Interactivo
- 4: Completado

La propiedad status mantiene el estado de la descarga en sí mismo. Representa los códigos estándar http que el navegador obtiene para la URL proporcionada.

Aquí se pueden ver parte de los posibles valores de la propiedad status (notar que el valor de 200 implica que todo ha ido bien):

- 200: OK
- 201: Creado
- 204: No hay contenido
- 205: Contenido reiniciado
- 206: Contenido parcial
- 400: Mala petición
- 401: No autorizado
- 403: Prohibido
- 404: No encontrado
- 405: Método no permitido
- 406: No aceptable
- 407: Autenticación de Proxy requerida
- 408: Tiempo de petición agotado
- 411: Longitud requerida
- 413: Entidad solicitada demasiado grande
- 414: URL solicitada demasiado larga
- 415: Tipo de medio no soportado
- 500: Error interno de servidor
- 501: No implementado
- 502: Mala pasarela
- 503: Servicio no disponible
- 504: Tiempo agotado de pasarela
- 505: Versión de http no soportada

Para asegurar que los datos se han descargado completamente y que todo ha ido bien, se comprobará que la propiedad **readyState** de **XMLHttpRequestObject** vale 4 y la propiedad status **vale 200 o vale 0.**

Para obtener los datos dentro del objeto **XMLHttpRequest**, se usará una de estas opciones:

- Si se obtienen datos que se quieren tratar como texto estándar, hay que usar la propiedad **responseText** del objeto
- Si los datos han sido formateados como XML, se usará la propiedad **responseXML**.

Ahora que ya se ha ajustado el código para manejar la respuesta del servidor cuando ha sido enviada toca conectar al servidor para obtenerla. Se usará el método **send** enviando un valor null para conectar con el servidor y solicitar los datos usando el objeto **XMLHttpRequest** que

ya se había configurado. Esa llamada a **send** es la que hace que se descarguen los datos ya que, una vez terminada, se lanza el manejador de evento que se ha incluido en **onreadystatechange**.

Como puede verse, la aplicación obtiene datos en segundo plano del servidor y los muestra sin ningún refresco de página.

A continuación, se pueden ver las distintas propiedades de este objeto:

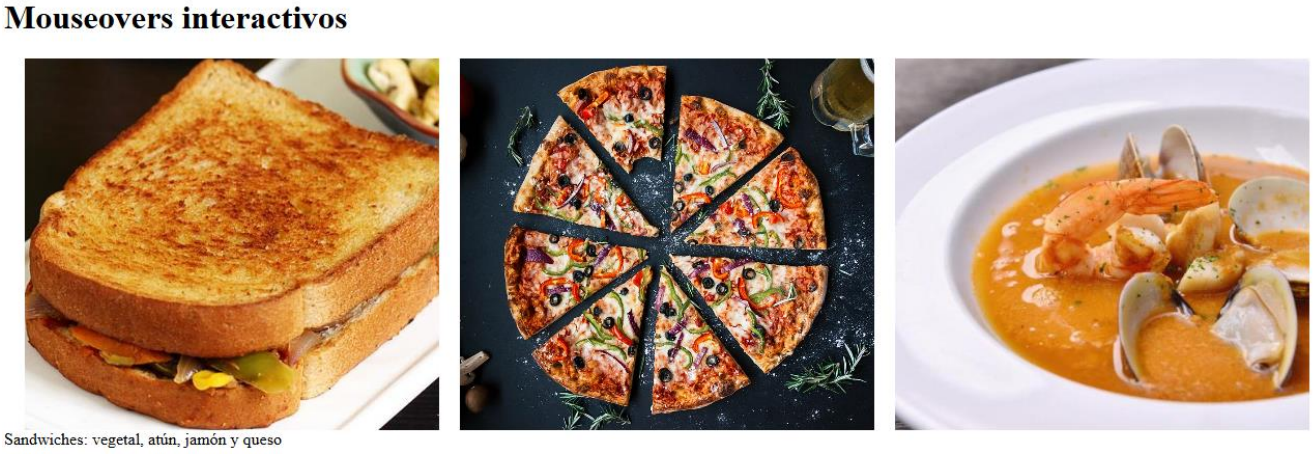
readyState	Guarda el estado de la petición (solo lectura)
responseText	Guarda la respuesta de body como una cadena (solo lectura)
responseXML	Guarda la respuesta body como XML (solo lectura)
status	Guarda el código de estado HTTP devuelto por una petición (solo lectura)
statusText	Guarda el texto de respuesta http (solo lectura)

En cuanto a los métodos:

abort	Aborta la petición http. Cuando se llama a este método, se reinicia el estado del objeto (readyState=0)
getAllResponseHeaders	Obtiene todos los encabezados HTTP
getResponseHeader	Obtiene el valor de un encabezado http
open	Abre una petición al servidor
send	Manda una petición HTTP al servidor
overrideMimeType	Sobrescribe el tipo MIME que devuelve el servidor

A continuación, se verá otro ejemplo Ajax que consiste en lo siguiente: al mover el ratón sobre la imagen en esta página, la aplicación trata el texto para ese mouseover usando Ajax. Esto no es difícil de implementar, todo lo que hay que hacer es conectar la función getData al evento onmouseover de cada una de las imágenes de la página.

El resultado sería similar al siguiente:



5. Aplicación Ajax con mouseover

El código del ejemplo:

```
<html>
  <head>
    <title>Probando Ajax</title>
    <script>
      … aquí iría el getData
    </script>
    <style>
      .imagenes { display: flex;
        justify-content: center;
        gap: 20px; }
      .imagenes img { width: 400px;
        height: auto;
        object-fit: cover; }
    </style>
  </head>
  <body>
    <h1>Mouseovers interactivos</h1>
    <div class="imagenes">
      <img src="sandwich.jpg" onmouseover="getData('sandwiches.txt', 'targetDiv')" />
      <img src="pizza.jpg" onmouseover="getData('pizzas.txt','targetDiv')" />
      <img src="sopa.jpg" onmouseover="getData('sopas.txt','targetDiv')" />
    </div>
    <div id="targetDiv">
      <p>Bienvenido a mi restaurante</p>
    </div>
  </body>
</html>
```

8.2.2 MÉTODOS HTTP

Las peticiones Ajax usan el protocolo HTTP (el mismo que utiliza el navegador para cargar una página). Este protocolo envía al servidor unas cabeceras HTTP (con información como el *userAgent* del navegador, el idioma, etc), el tipo de petición y, opcionalmente, datos o parámetros (por ejemplo, en la petición que procesa un formulario se envían los datos del mismo).

Existen diferentes tipos de petición que se pueden hacer:

- **GET**: suele usarse para obtener datos sin modificar nada (equivale a un SELECT en SQL). Si se envían datos (ej. la ID del registro a obtener) suelen ir en la url de la petición (formato URIEncoded).
  Ejemplos de uso:
  - http://locahost/users/3
  - https://jsonplaceholder.typicode.com/users
  - www.google.es?search=js

- **POST**: suele usarse para añadir un dato en el servidor (equivalente a un INSERT). Los datos enviados van en el cuerpo de la petición HTTP (igual que sucede al enviar desde el navegador un formulario por POST)
- **PUT**: es similar al *POST* pero suele usarse para actualizar datos del servidor (como un UPDATE de SQL). Los datos se envían en el cuerpo de la petición (como en el POST) y la información para identificar el objeto a modificar en la url (como en el GET). El servidor hará un UPDATE sustituyendo el objeto actual por el que se le pasa como parámetro
- **PATCH**: es similar al PUT pero la diferencia es que en el PUT hay que pasar todos los campos del objeto a modificar (los campos no pasados se eliminan del objeto) mientras que en el PATCH sólo se pasan los campos que se quieren cambiar y el resto permanecen como están
- **DELETE**: se usa para eliminar un dato del servidor (como un DELETE de SQL). La información para identificar el objeto a eliminar se envía en la url (como en el GET)

Además, existen otros tipos que no se verán aquí (como *HEAD*, *PATCH*, etc)

El servidor acepta la petición, la procesa y le envía una respuesta al cliente con el recurso solicitado y además unas cabeceras de respuesta (con el tipo de contenido enviado, el idioma, etc) y el código de estado. Los códigos de estado más comunes son:

- 2xx: son peticiones procesadas correctamente. Las más habituales son 200 (*ok*) o 201 (*created*, como respuesta a una petición POST satisfactoria)
- 3xx: son códigos de redirección que indican que la petición se redirecciona a otro recurso del servidor, como 301 (el recurso se ha movido permanentemente a otra URL) o 304 (el recurso no ha cambiado desde la última petición por lo que se puede recuperar desde la caché)

- 4xx: indican un error por parte del cliente, como 404 (*Not found*, no existe el recurso solicitado) o 401 (*Not authorized*, el cliente no está autorizado a acceder al recurso solicitado)
- 5xx: indican un error por parte del servidor, como 500 (error interno del servidor) o 504 (*timeout*, el servidor no responde).

En cuanto a la información enviada por el servidor al cliente normalmente serán datos en formato **JSON** o XML (cada vez menos usado) que el cliente procesará y mostrará en la página al usuario. También podría ser HTML, texto plano, etc.

Para convertir objetos en cadenas de texto *JSON* y viceversa JavaScript proporciona 2 funciones:

- **JSON.stringify(*objeto*)**: recibe un objeto JS y devuelve la cadena de texto correspondiente. Ej.: const cadenaAlumnos = JSON.stringify(alumnos)
- **JSON.parse(*cadena*)**: realiza el proceso inverso, convirtiendo una cadena de texto en un objeto. Ej.: const alumnos = JSON.parse(cadenaAlumnos)

8.2.3 FUNCIONES CALLBACK

Las funciones callback son aquellas que se pasan como argumentos en otras funciones para que sean ejecutadas en un punto posterior en el tiempo. A continuación, un ejemplo de uso:

```
// Multiplica dos números
function multiplicar(x, y) { return x * y; }
// Suma dos números
function sumar(x, y) {
  return x + y;
}
// Usa un callback para procesar dos números
function calcular(x, y, operar) {
  return operar(x, y);
}
let a = calcular(10, 5, sumar); // usa el callback sumar
console.log(a); // muestra 15
let b = calcular(10, 5, multiplicar); // usa el callback multiplicar
console.log(b); // muestra 50
```

Además, los callbacks se pueden crear y usar sin atarlos a un nombre específico de función, es decir, como anónimos. Principalmente, son útiles cuando solo se necesitan declarar una vez ya que son más rápidos de escribir que los que tienen nombre. Un ejemplo de código de esto podría ser el siguiente:

```
let d = calcular(10, 5, (x, y) => x - y);
console.log(d); // muestra 5
```

Funciones como **map()** y **filter()** vistas anteriormente usan funciones callback.

Existe un estilo de programación llamado Continuation Passing Style (CPS) o estilo de paso de continuación que se basa en encadenar funciones callback juntas. En CPS, los métodos con callbacks como argumentos se llaman dentro de otras funciones callbacks. Se caracteriza por tener métodos que tienen funciones callback como su último argumento. Un ejemplo:

```
function miFuncion(x, callback) {  callback(x); }
let respuesta = 0;
miFuncion(10, function (x) {
  //callback1
  let result = x * x; //result = 100
  miFuncion(result, function (x) {
    //callback2 dentro de callback 1
    var result2 = x + x; //result2 = 200
    miFuncion(result2, function (x) {
      //callback 3 dentro de callback 2
      respuesta = x + 100;
      console.log(respuesta); // muestra 300
    });
  });
});
```

CPS tiende a ser más difícil de manejar a medida que se van encadenando más callback como cabría de esperar.

8.2.4 PROMESAS

Las promesas son esencialmente contenedores para valores que todavía no están disponibles, pero podrían llegar a estarlo eventualmente. Su importancia radica en que están siendo la vía estándar de manejar funciones asíncronas en JavaScript.

El siguiente código ilustra la creación de una nueva promesa:

```
let promesa = new Promise(function (resolve, reject) {
  // Se hacen cosas
  let exitosa = true;
  if (exitosa) {
    resolve("¡Todo ha salido bien!"); /* Si todo ha salido bien */
  } else {
    reject(Error("¡Algo ha salido mal!")); /* Si algo ha salido mal */
  }
});
```

El constructor **new Promise()** se llama para crear una nueva promesa y lo hace tomando una función **callback** con los argumentos **resolve** y **reject**. La función **resolve** se usa para cambiar el estado de la promesa de pendiente a completada. El valor que se le pasa a esa función se convierte en el valor de cumplimiento de la promesa. Una vez que se llama a **resolve**, otras posibles futuras llamadas a **resolve** o **reject** no van a tener ningún efecto. Viendo el código, hay que darse cuenta de que cuando se usa el método **resolve** se le da el valor que tendrá la promesa al ser completada.

Por otra parte, la función **reject()** se usa para cambiar el estado de la promesa de pendiente a rechazada. De forma similar a **resolve**, el valor pasado va a ser el correspondiente al rechazo de la promesa; e igualmente, una vez llamada, ya no se atenderán otras de tipo resolve o reject. Lo normal en estos casos es pasarle un objeto Error ya que proporciona un informe más detallado de la incidencia.

Promise.resolve() se usa para devolver una promesa que ya ha sido completada. Por otra parte, **Promise.reject()** se puede usar para devolver una promesa ya rechazada. Ambos métodos se llaman fuera del constructor new Promise():

```
// Promesa resuelta con el valor de cumplimiento "ya resuelta"
let promesaResuelta = Promise.resolve("ya resuelta");
```

Promise.reject() se usa para crear una promesa ya rechazada:

```
// Promesa rechazada con el valor de rechazo "ya rechazada"
let promesaRechazada = Promise.reject("ya rechazada");
```

Si se pasa como argumento otra promesa a **resolve()** entonces, la nueva promesa toma el valor de cumplimiento de la que se ha pasado:

```
let primeraPromesa = Promise.resolve("ya resuelta");
// El valor de cumplimiento de segundaPromesa sería "ya resuelta"
let segundaPromesa = Promise.resolve(primeraPromesa);
```

Los métodos **then()** y **catch()** se usan para manejar los resultados de promesas una vez que han terminado de estar pendientes. El método **then()** se usa para manejar promesas resueltas mientras que catch() se usa para las rechazadas. Ambos usan funciones callback y cada una de ellas debería tener un argumento que represente el resultado de la promesa.

Se muestra un ejemplo de código con lo visto hasta ahora:

```
let promesa = new Promise(function(resolve, reject) {

    // Se hacen cosas
    let exito = true;

    setTimeout(function(){ //procesamiento asíncrono  después de 5 segundos
      if (exito) { // si todo ha salido bien
          resolve('Todo ha salido bien');
      }
      else{ // Si algo ha salido mal
          reject(Error("Algo ha salido mal"))
      }
    },5000);
});

// El estado de la promesa cambia de pendiente a resuelta después de 5 segundos
promesa.then(function(val){//val representa el valor de cumplimiento
    console.log(val);//muestra "Todo ha salido bien" ya que la promesa se ha resuelto
}).catch(function(val){//val representa el valor de rechazo
    console.log(val); //no ocurre ya que la promesa nunca se rechaza
});
```

El método then() puede ser llamado mediante un callback de éxito y otro de fallo como alternativas al uso de los métodos then y catch:

```
promesa.then(function(val){// Callback de éxito
    console.log(val);
},function(val){// Callback de rechazo
    console.log(val);
})
```

Los resultados de promesas se pueden transformar llamando al comando **return** dentro del **then().**

Esto provocará que then devuelva una nueva promesa con el resultado transformado:

```
let promesa = Promise.resolve("hola");

let promesa2 = promesa.then(function(resultado) {
    console.log(resultado) //Muestra "hola"
    return resultado + " mundo" // agrega " mundo" al resultado y establece esto como
el nuevo valor de cumplimiento de promesa2
});

promise2.then(function(result){
    console.log(result); //Muestra "hola mundo"
});
```

Además, se pueden encadenar varias transformaciones juntas usando múltiples llamadas a then. En el código a continuación se puede ver un ejemplo de llamadas encadenadas:

```
let promesa = Promise.resolve([1,2,3,4]);

promesa.then(function(resultado) {
    console.log(result) //Muestra [1,2,3,4]
    // Calcula el cuadrado de cada valor en el arraysquares each value in the array
    return result.map(x => x * x);
}).then(function(result2){
    console.log(result2) //Muestra [1,4,9,16]
    //Filtra dejando solo los elementos mayores que 10
    return result2.filter( x => x > 10);
}).then(function(result3){
    console.log(result3) //Muestra [16]
    //Convierte result3 a una cadena de caracteres y agrega "!!"
    return result3.toString() + "!!";
}).then(function(result4){
    console.log(result4) //Muestra "16!!"
    //Devuelve una promesa con "16!!" como valor de cumplimiento
    return result4;
}).catch(function(error){
    console.log(error)
});
```

8.2.4.1 Cadena de promesas vs pirámide de Callbacks

En este apartado se comparará el uso de promesas con el de CPS cuando se intentan encadenar operaciones asíncronas.

En este primer ejemplo, se muestra el encadenado de callbacks:

```
obtenerNumeroAleatorio(function(numero) {
    console.log(numero); // por ejemplo, muestra 42
    obtenerNombreDesdeNumero(numero, function(nombre) {
        console.log(nombre) // por ejemplo, muestra 'Bob'
        obtenerEdadDesdeNumero(nombre, function(edad) {
            console.log(edad) // por ejemplo, muestra 21
            // Se hace algo con edad
        },
        function(error) {
            console.log(error);
        });
    },
    function(error) {
        console.log(error);
    });
}, function(error) {
    console.log(error);
});
```

Y en este otro, con promesas:

```
// Devuelve una promesa que contiene un número aleatorio
obtenerNumeroAleatorio().then(function(resultado) {
    console.log(resultado) // p.e. 42
    return obtenerNombreDesdeNumero(resultado); //devuelve una promesa conteniendo
una representación en cadena de un nombre
}).then(function(result2){
    console.log(result2) // p.e. "Bob"
    return obtenerEdadDesdeNombre(result2);  //devuelve una promesa conteniendo un
número que representa una edad
}).then(function(result3){
    console.log(result3) // p.e. 21
}).catch(function(error){
    console.log(error)
});
```

Como puede verse, es difícil hacer cambios para encadenar operaciones asíncronas usando CPS, especialmente si hay un callback para éxito y otro para fallo. Las promesas permiten encadenar de forma más escalable operaciones asíncronas.

8.2.4.2 Promise.all()

El método **Promise.all()** se usa para procesar varias promesas al mismo tiempo. Toma un arrray de promesas y a continuación, espera a que se resuelvan todas. Una vez que han terminado de resolverse, se obtiene un array de resultados mediante el método **then().** Si alguna de ellas se rechaza, devolverá la primera promesa rechazada.

Un ejemplo de uso podría ser el siguiente:

```
let promesa1 = Promise.resolve('hola');
let promesa2 = Promise.resolve({edad:2,altura:188})
let promesa3= 42;
Promise.all([promesa1,promesa2,promesa3]).then(function(result) {
    console.log(result) // Muestra el array ["hola",{edad:2,altura:188},42]
}).catch(function(error){
    console.log(error)
});
```

En esta porción de código la llamada a **Promise.all()** llama a rejects cuando se rechaza una de las promesas:

```
let promesa1 = Promise.resolve("hola");
let promesa2 = Promise.resolve({ edad: 2, altura: 188 });
let promesa3 = Promise.reject("error");
Promise.all([promesa1, promesa2, promesa3])
  .then(function (result) {
    console.log(result); // No hace nada ya que se ha rechazado promesa3
  })
  .catch(function (error) {
    console.log(error); // Muestra el error
});
```

8.2.4.3 Promise.race()

El método **Promise.race()** toma un array de promesas y dará como resultado la promesa que se rechace o resuelva más rápido. Como en cualquier promesa, los métodos **then()** y **catch()** se usan para devolver los resultados de la promesa más rápida. El método **Promise.race()** se puede usar para elegir la fuente más rápida cuando haya dos fuentes similares de los mismos datos:

```
let promesa1 = new Promise(function(resolve,reject){
    setTimeout(function(){
        resolve("Terminado en dos segundos");
    },2000) // devuelve una promesa resuelta después de 2 segundos
});
let promesa2 = new Promise(function(resolve,reject){
    setTimeout(function(){
        resolve("Terminado en cinco segundos");
    },5000) // devuelve una promesa resuelta después de 5 segundos
});
Promise.race([promesa1,promesa2]).then(function(result) {
    console.log(result) // muestra "Terminado en dos segundos" porque promesa1 se
resuelve primero
}).catch(function(error){
    console.log(error)
});
```

El método **Promise.race()** también se puede usar para limitar la cantidad de tiempo que las promesas deben tardar en resolver incluyendo una que fuerce a rechazar después de un tiempo dado.

Un ejemplo:

```
let promesaResuelve10Segundos = new Promise(function (resolve, reject) {
  setTimeout(function () {
    resolve("Terminada en diez segundos");
  }, 10000); // Devuelve una promesa resuelta después de 10 segundos
});

let promesaResuelve5Segundos = new Promise(function (resolve, reject) {
  setTimeout(function () {
    reject("Error: La promesa ha tardado más de 5 segundos en resolver");
  }, 5000); // devuelve una promesa rechazada después de 5 segundos
});

Promise.race([promesaResuelve10Segundos, promesaResuelve5Segundos])
  .then(function (result) {
    console.log(result); // No ocurre porque se rechaza promesaResuelve5Segundos
  })
  .catch(function (error) {
    console.log(error); // Muestra el error "Error: La promesa ha tardado más de 5
segundos en resolver"
});
```

8.2.5 LA API FETCH

La API Fetch es un interfaz que se usa para hacer peticiones en red, generalmente a un servidor. Es una mejora muy importante sobre **XMLHttpRequest**; además se ha construido para navegadores modernos y también devuelve promesas.

A continuación, se puede ver un uso básico de fetch:

```
fetch("https://jsonplaceholder.org/posts/1")
  .then((resultado) => resultado.json())
  .then((resultado) => {
    console.log(resultado); // Se muestra el objeto una vez completada la llamada
  })
  .catch((err) => {
    console.log(err);
});
```

En este se llama a una URL que devuelve un JSON y muestra el objeto obtenido (resultado) a partir de él en la consola.

fetch(url). El método fetch() toma un endpoint en forma de URL y lo usa para hacer una petición en red; además, devuelve una promesa (Promise) que contiene un objeto **Response**:

```
fetch("https://jsonplaceholder.org/posts/1") //fetch() recoge un endpoint como URL
 .then((resultado) => { //resultado contiene un objeto Response
});
```

Extrayendo datos de un objeto Response

Un objeto Response tiene varios métodos que se usan para extraer los datos obtenidos:
json() se usa para extraer un objeto json.
text() se usa para extraer una cadena de texto
blob() se usa para extraer un objeto de tipo fichero

El ejemplo visto en primer lugar obtiene el resultado mediante json(). Su tipo es JSONobject. En este otro ejemplo, la respuesta es en forma de texto:

```
fetch("https://jsonplaceholder.org/todos/1")
  .then((resultado) => resultado.text())
  .then((resultado) => {
     console.log(resultado); // Se muestra el objeto una vez completada la llamada
});
```

En este caso, devolverá el JSON como texto plano.

Comprobando el estado de Response

Es importante comprobar el estado del objeto Response obtenido. Los estados o status son los mismos que ya se vieron anteriormente cuando se habló del XMLHttpRequest. En el siguiente ejemplo se muestra cómo se gestiona un estado no válido (p.e. 404) antes de manejar la respuesta definitiva:

```
// fetch de una URL no válida
fetch("https://jsonplaceholder.org/bad_url/1")
  .then((resultado) => {
     console.log(resultado);
     // resultado.ok se da cuando la respuesta está entre 200 y 299
     if(resultado.ok) return resultado.text();
     else{
        console.log(resultado.status); // mostrará 404 (no encontrada)
        return Promise.reject(resultado.status)
     }
   })
   .then((resultado) => {
     console.log(resultado);
    })
   .catch((err) => {
     console.log("Error: " + err);}
);
```

Init object en fetch

El método fetch() también puede tomar opcionalmente un objeto de inicialización (init object en inglés).

Este objeto se puede pasar en la llamada de la propia función:

```
let initObject = {
        method: "POST",
        headers: new Headers(),
        mode: "cors",
        body: "{}",
};

// El initObject especifica método, encabezados, modo y cuerpo de la petición
fetch("https://jsonplaceholder.org/posts", initObject)
        .then((resultado) => resultado.json())
        .then((resultado) => {
          console.log(resultado);
        })
        .catch((err) => {
          console.log("Error: " + err);
});
```

El método devuelve objeto de id 101, que corresponde con el próximo id disponible en la lista de objetos que contiene el JSON de la petición. A continuación, se verán los distintos parámetros de los que consta el initObject:

- method. Cadena que se usa para especificar el tipo de método de petición HTTP (apartado 8.2.2)
- body. Se trata de una cadena JSON que se usa para enviar datos mediante la petición fetch. Si su valor es un objeto, es importante aplicar JSON.stringify() sobre él o no se procesará correctamente. Las peticiones GET y HEAD no pueden tener este atributo body. Un ejemplo de objeto a pasar por body:

```
let bdy = {
    id: 12345,
    nombre: 'abc',
    edad: 21
}
let initObject = {
    body: JSON.stringify(bdy)
}
```

- headers: Se usa para añadir más información sobre el recurso que está siendo enviado o sobre el cliente. Este objeto se crea mediante el constructor new Headers() y los encabezados individuales se añaden al objeto usando el método append(). Ejemplo:

```
let encabezados = new Headers();
encabezados.append('Content-Type', 'application/json');
let initObject = {
    headers: encabezados
}
```

- mode. Se trata de una cadena que se usa para determinar si la petición puede obtener recursos de distintos servidores. Principalmente, hay dos tipos:
  - same-origin. La petición solo puede traer recursos del mismo servidor
  - cors (cross origin). La petición puede traer recursos de distintos servidores

Utilizando Fetch con Requests

El método fetch() puede tomar un objeto Request en lugar de una URL y un init object. El constructor Request toma los mismos parámetros que el método fetch(): una URL y un objeto opcional initObject. Los objetos Request se usan porque hacen que la petición sea un poco más clara y ofrecen algo más de control. Un ejemplo de uso:

```
let initObject = {
      method: "POST",
      headers: new Headers(),
      mode: "cors",
      body: "{}",
};
// Crea un objeto request usando URL e initObject
let request = new Request("https://jsonplaceholder.typicode.com/posts",initObject);
fetch(request)
        .then((resultado) => resultado.json())
        .then((resultado) => {
          console.log(resultado);
        })
        .catch((err) => {
          console.log("Error: " + err);
});
```

Reutilizando objetos Request

En este caso, hay que hacer dos grupos: las peticiones con cuerpo (body) y las que no lo incluyen. Si un objeto Request se usa más de una vez en una petición Fetch que involucra un elemento body (POST, PUT), lanzará un error.

```
let initObject = { method: "POST", headers: new Headers(), mode: "cors", body: "{}"};
// Crea un objeto request usando URL e initObject
let request = new Request("https://jsonplaceholder.typicode.com/posts",initObject);
fetch(request)
  .then((resultado) => resultado.json())
  .then((resultado) => {
    console.log(resultado);
  })
  .catch((err) => {
    console.log("Error: " + err);});
// Se lanza la petición por segunda vez
fetch(request)
  .then((resultado) => resultado.json())
  .catch((err) => {
    console.log("Error: " + err.message);});
```

El sistema mostrará el siguiente error: "Error: Failed to execute 'fetch' on 'Window': Cannot construct a Request with a Request object that has already been used.". Esto no pasa si la petición es de tipo GET o HEAD ya que no requieren un elemento **body**:

```
let request = new Request("https://jsonplaceholder.typicode.com/posts/1");
// Se lanza una primera petición
fetch(request).then((resultado) => {
  console.log(resultado.status);
});
// Reutilizando la petición
fetch(request).then((resultado) => {
  console.log(resultado.status);
});
```

En ambos casos, la consola muestra un estado 200, es decir, una petición correcta.

8.2.5 ASYNC / AWAIT

Async / Await es una característica moderna de JavaScript que simplifica el trabajo con promesas haciendo que el código asíncrono parezca como síncrono. Un ejemplo de uso podría ser el siguiente:

```
const obtenerDatosConAsyncAwait = async () => {
  try {
    const data = await obtenerDatos();
    console.log("Datos obtenidos exitosamente con Async/Await:", data);
  }
  catch (error) {
    console.error("Error obteniendo datos con Async/Await:", error);
  }
};
```

A continuación, se muestra cómo se pueden manejar excepciones con bloques try/catch con Async / Await asegurando un manejo de potenciales excepciones mucho más suave. Además de la función obtenerDatosConAsyncAwait se incluye el siguiente código antes de dicho método:

```
const obtenerDatos = () => (new Promise((resolve, reject) => {
    setTimeout(() => {
      const error = true; // Se simula un error
      if (error) {
        reject("Error: ha fallado la obtención de datos");
      } else {
        const datos = [1, 2, 3, 4, 5];
        resolve(datos);
      }
    }, 2000);
  })
)
```

Por último, se realiza la llamada a la función **obtenerDatosConAsyncAwait**:

```
obtenerDatosConAsyncAwait();
```

En el ejemplo se tiene una función **obtenerDatos** que devuelve una promesa. Dentro del callback de la promesa se simula un error. En ese caso, se rechaza también la promesa con un mensaje de error. De otra forma, se resolvería la promesa con una serie de datos.

La función **obtenerDatosConAsyncAway** se define con la palabra clave **async**, que permite el uso de **await** dentro de la función. En ella se usa un bloque try-catch para manejar posibles excepciones.

Cuando se llama al método **obtenerDatosConAsyncAway**, se ejecuta el código dentro de la función. Si ocurre un error durante la operación asíncrona dentro de **obtenerDatos**, se capturará en el bloque catch y se mandará el mensaje a consola. De otra forma, se obtienen los datos exitosamente mandándolos a consola.

Async / Await se puede combinar con fetch para obtener datos de un servidor:

```
let request = new Request("https://jsonplaceholder.org/posts/1");
const obtenerDatos = async () => {
  try {
    const respuesta = await fetch(request);
    const datos = await respuesta.json();
    console.log(datos);
  } catch (error) {
    console.log("Error obteniendo datos: ", error);
  }
};
```

Ventajas de Async/Await:

- Se mejora la legibilidad del código reduciendo el infierno de los callbacks
- Manejo de errores simplificado. El flujo de errores en promesas y Async / Await hacen más fácil capturar y manejar errores en operaciones asíncronas
- Encadenado de múltiples operaciones. Ambas técnicas permiten encadenar múltiples operaciones asíncronas de forma eficiente mejorando la organización del código.

Errores comunes de async / await

Aunque async/await ofrece muchos beneficios, hay que tener cuidado con ciertos errores muy comunes que pueden provocar malas experiencias:
1. Olvidarse del await. Sin await, la función async devuelve una promesa que no ha sido resuelta aún. Esto puede llevar a un comportamiento inesperado si se está intentando acceder a datos de la promesa.

En el siguiente código se ilustra este fallo (spoiler: mostrará un error de conexión):

```
const obtenerDatos = async () => {
        const datos = fetch("https://jsonplaceholder.org/posts/1");
        console.log(datos);
};
```

2. Bloquear código con múltiples awaits. Si se tienen múltiples tareas asíncronas esperando que vayan llegando una por una pueden bloquear el hilo principal innecesariamente. En lugar de esto, se pueden ejecutar en paralelo con **Promise.all()**.

```
const obtenerDatos = async () => {
        const [usuario, posts] = await Promise.all([
            obtenerDatosUsuario(),
            obtenerDatosPosts()
        ]);
        console.log(usuario, posts);
};
```

3. No manejar rechazos propiamente. Utilizar siempre try/catch para manejar rechazos dentro de una función async. Si una promesa se rechaza y no hay manejo de excepciones, la promesa quedará sin gestionar, expuesta a bugs.

```
const obtenerDatos = async () => {
        try {
          const respuesta = await fetch(
            "https://jsonplaceholder.org/posts/1"
          );
          const datos = await respuesta.json();
          console.log(datos);
        } catch (error) {
          console.error("Error en la obtención:", error);
        }
};
```

Trucos avanzados manejando async / await

Usar await dentro de bucles: Si se necesita esperar por algo dentro de un bucle, es una buena práctica asegurar que se hace un await en cada iteración para evitar bloquear el bucle innecesariamente.

```
const procesarItems = async (items) => {
        for (let item of items)
          await procesarItem(item);

};
```

Refactorizar funciones largas. Si la función async llega a ser muy larga, considerar refactorizarla en algo más pequeño, en funciones más manejables para mejorar la legibilidad y ser más fáciles de probar.

8.3 PROGRAMACIÓN DE APLICACIONES CON COMUNICACIÓN ASÍNCRONA. MODIFICACIÓN DINÁMICA DEL DOCUMENTO

8.3.1 UTILIZACIÓN DE XMLHTTPREQUEST CON XML

Como su propio nombre indica, Ajax es Asynchronous Javascript and XML, así que en este ejemplo se va a ver cómo puede participar XML en todo esto. En el ejemplo siguiente, se le dan al usuario varias opciones para ajustar el color del texto en la página Web. Se construirá una aplicación con un desplegable que contenga varios colores que serán obtenidos usando XMLHttpRequest y datos formateados como XML. Esta aplicación tiene dos esquemas de color:

- Esquema 1:
  - Rojo (red)
  - Verde (green)
  - Azul (blue)
- Esquema 2:
  - Negro (black)
  - Blanco (white)
  - Naranja (orange)

El usuario puede seleccionar estos dos esquemas simplemente haciendo clic en los botones para tal efecto. Cuando hace clic en un botón, los colores correspondientes al esquema son cargados en el desplegable. Al seleccionar un color, automáticamente cambia el color del texto.

Para la aplicación de ejemplo, un script PHP proporciona los colores en cada esquema (**opciones.php**). Este script devuelve sus datos usando XML y recoge el esquema mediante un parámetro GET. El código a continuación es el que manda opciones.php?esquema=1 al navegador:

```
<?xml version="1.0"?>
<opciones>
    <opcion>red</opcion>
    <opcion>green</opcion>
    <opcion>blue</opcion>
</opciones>
```

El elemento <opciones> encierra tres elementos <opcion>, los cuales contienen texto correspondiente a un color: rojo, verde y azul. ¿Cómo se puede devolver el XML usando un script PHP?. Lo primero que hay que hacer es ajustar el encabezado de tipo de contenido a <text/xml>. Esto informa al navegador que los datos que se están creando son XML y deben ser tratados como tal.

Nota: se puede montar un servidor PHP básico mediante la orden **php -S localhost:8000** en el directorio en el que se tengan los archivos. Hay que tener en cuenta que php como comando solo puede lanzarse si está en el path del sistema operativo. Si no, hay que poner la ruta absoluta (por ejemplo, en Windows, C:\php\php.exe…).

El código PHP que dará esa salida es el siguiente (opciones.php):

```php
<?php
header('Access-Control-Allow-Origin: *');
header("Content-Type: application/xml; charset=UTF-8");

$esquema = filter_input(INPUT_GET, 'esquema', FILTER_VALIDATE_INT);
if ($esquema === null) {
    http_response_code(400);
    die("No se ha proporcionado un esquema");
}
switch ($esquema) {
    case 1:
        $opciones = ['red', 'green', 'blue'];
        break;
    case 2:
        $opciones = ['black', 'white', 'orange'];
        break;
    default:
        http_response_code(400);
        die("El esquema proporcionado no es válido");
}

$xml = new SimpleXMLElement('<opciones/>');
foreach ($opciones as $opcion)
    $xml->addChild('opcion', htmlspecialchars($opcion));
echo $xml->asXML();
?>
```

Ahora viene la parte importante de la aplicación. Entra en acción opciones.html. Dos botones permiten al usuario seleccionar entre dos esquemas y esos dos botones llaman a la función **getOpcion** con el número de esquema. Como la respuesta del servidor viene como XML, se va a leer con la propiedad **responseXML** del objeto **XMLHttpRequest**, el cual se devuelve en una variable llamada **xmlDoc**. ¿Como se puede manejar esto en JavaScript? Esto no es complicado, se puede usar **getElementsByTagName** para obtener todos los elementos de un cierto nombre. En este caso, los elementos de los cuales se necesitan sus datos son los <opcion>, así que se obtendrán y almacenarán en una variable opciones.

Para extraer información de XML, este ejemplo llama a otra función listaOpciones, la cual desempaqueta los colores y los almacena en la lista desplegable. Los datos se almacenan como un array de elementos <opcion> el cual facilita las cosas puesto que solo hay que recorrerlo. Mediante **opciones[i].firstChild.data** se recupera el texto de ese nodo que sirve para dar además el estilo al texto. Para añadir los colores al control <select> usamos la propiedad options, la cual guarda los objetos listados en el control. Cada item que se quiera añadir a la lista es un objeto option. Así la lista desplegable mostrará los colores disponibles para el esquema de color que el usuario haya escogido.

El código completo es el siguiente:

```
<html>
  <head>
    <script>
      let opciones;
      const listaOpciones = (opciones) => {
        let cbOpciones = document.getElementById("cbOpciones");
        for (let i = 0; i < opciones.length; i++)
          cbOpciones.options[i] = new Option(opciones[i].firstChild.data);
      };
      const getOpcion = (opc) => {
        let XMLHttpRequestObject = new XMLHttpRequest();
        url = `http://localhost:8000/opciones.php?esquema=${opc}`;
        event.preventDefault();
        if (XMLHttpRequestObject) {
          XMLHttpRequestObject.open("GET", url);
          XMLHttpRequestObject.onreadystatechange = () => {
            if (XMLHttpRequestObject.readyState == 4) {
              if (XMLHttpRequestObject.status == 200) {
                let xmlDoc = XMLHttpRequestObject.responseXML;
                opciones = xmlDoc.getElementsByTagName("opcion");
                listaOpciones(opciones);
              }
            }
          };
          XMLHttpRequestObject.send(null);
        }
      };
      const setOpcion = () => {
        document.getElementById("targetDiv").style.color =
          opciones[
            document.getElementById("cbOpciones").selectedIndex
          ].firstChild.data;
      };
    </script>
  </head>
  <body>
    <h1>Usando Ajax y XML</h1>
    <form>
      <select size="1" id="cbOpciones" onchange="setOpcion()">
        <option>Selecciona un esquema</option>
      </select>
      <button onclick="getOpcion(1)">Usar esquema 1</button>
      <button onclick="getOpcion(2)">Usar esquema 2</button>
    </form>
    <div id="targetDiv" width="100" height="100">Aplica color a este texto</div>
  </body>
</html>
```

En el ejemplo se ha usado GET para pasar los datos, pero podría ser perfectamente con POST con solo cambiar una línea (aunque lo suyo es añadirle todas las comprobaciones previas):

```
$esquema = filter_input(INPUT_POST, 'esquema', FILTER_VALIDATE_INT);
```

También hay que cambiar algunas líneas en JavaScript, como la de la petición:

```
XMLHttpRequestObject.open("POST", url);
```

Y añadir la siguiente:

```
XMLHttpRequestObject.setRequestHeader("Content-Type", "application/x-www-form-
urlencoded");
```

Aquí lo fundamental es que ya no se enviará la petición con null, sino dándole el esquema:

```
XMLHttpRequestObject.send(`esquema=${opc}`);
```

8.3.2 UTILIZACIÓN DE PROMESAS. CATÁLOGO DE PRODUCTOS

En este ejemplo se va a ilustrar el uso de promesas como mecanismos de comunicación asíncrona vistos hasta ahora. Fundamentalmente, tendrá los tres siguientes ficheros:

- catalogoProductos.html – La página de productos
- catalogoProductos.js – La parte de JavaScript de la página
- lib.js – Biblioteca JavaScript usada para proporcionar información asíncrona mediante promesas

8.3.2.1 Biblioteca JavaScript

Primeramente, se construirá la biblioteca lib.js usada para proporcionar información de productos de forma asíncrona mediante promesas. La biblioteca tendrá las siguientes funciones:

- obtieneProductoPorId(id): Devuelve una promesa conteniendo el producto con el ID especificado
- obtieneProductosPorTipo(tipo): Devuelve una promesa conteniendo un array de productos con el tipo especificado
- obtieneProductosPorPrecio(precio, intervalo): Devuelve una promesa conteniendo un array de todos los productos que están dentro de una cierto intervalo con respecto al precio especificado
- obtieneProductos(): Devuelve una promesa conteniendo un array de todos los productos en el catálogo
- creaObjetoAleatorio(): Crea un producto aleatorio (no es accesible fuera del fichero de biblioteca lib.js)
- creaCatalogoAleatorio(): Crea un catálogo de productos aleatorios (no accesible fuera del fichero de biblioteca lib.js)

Para comenzar, se añade el siguiente código a lib.js:

```
(function (window) {
  const miBiblioteca = () => {
    // Las definiciones de función van aquí
    // El código de ejecución va aquí
    return {
      obtieneProductoPorId: obtieneProductoPorId,
      obtieneProductosPorPrecio: obtieneProductosPorPrecio,
      obtieneProductosPorTipo: obtieneProductosPorTipo,
      obtieneProductos: obtieneProductos,
    };
  }
  if (typeof window.api === "undefined") {
    window.api = miBiblioteca();
  }
})(window);
```

Este código creará un objeto que tendrá una pareja de funciones de biblioteca como atributos. Las asignará al objeto global api. Después de incluir este fichero lib.js, las funciones de biblioteca se podrán acceder a través de la función global api.
Algunos ejemplos (no introducir en el proyecto):

```
let promesa1 = api.obtieneProductos();
let promesa2 = api.obtieneProductoPorId(42);
let promesa3 = api.obtieneProductosPorPrecio(200,25);
let promesa4 = api.obtieneProductosPorTipo("Libro");
```

En el mismo fichero, a partir de "Las definiciones de función van aquí" se añade la definición de **creaObjetoAleatorio** que creará un producto con un tipo y precio aleatorios. El tipo de producto puede ser Electrónica, Libro, Ropa o Comida. El precio de producto oscilará entre 0 y 500:

```
const creaObjetoAleatorio = () => {
    const arrayTipos = ['Electrónica','Libro','Ropa','Comida'];
    const precio = (Math.random()*500).toFixed(2)
    const tipo = arrayTipos[Math.floor(Math.random()*4)];
    return {precio:precio, tipo:tipo};
};
```

A continuación, **creaCatalogoAleatorio** la cual devolverá un array conteniendo un número concreto de productos aleatorios. Cada uno de ellos tendrá un id, precio y tipo.

```
const creaCatalogoAleatorio = (num) => {
  const catalogo = [];
  for (let i = 0; i < num; i++) {
    const obj = creaObjetoAleatorio();
    catalogo.push({ id: i, precio: obj.precio, tipo: obj.tipo });
  }
  return catalogo;
};
```

El código de llamada a la función anterior irá justo a continuación del comentario "El código de ejecución va aquí":

```
const catalogo = creaCatalogoAleatorio(100);
```

La siguiente función tiene como objetivo devolver una promesa con la lista completa de productos. Esta promesa se resolverá un segundo después de que se ejecute la función. Este código se coloca en la sección de definición de funciones:

```
const obtieneProductos = () => {
    const promesa = new Promise(function(resolve, reject) {
        setTimeout(function(){
            resolve(catalogo);
        },1000);
    });
    return promesa;
};
```

La función **obtieneProductoPorId(id)** buscará dentro del catálogo devolviendo una promesa que contenga el producto que coincida con el id dado. La promesa también se resolverá en un segundo después de que la función se ejecute. Se rechazará si se proporciona una id no válida. A continuación, el código de búsqueda por id:

```
const obtieneProductoPorId = (id) => {
      const promesa = new Promise(function (resolve, reject) {
        let i = 0;
        setTimeout(function () {
          const producto = catalogo.find((p) => p.id == id);
          if (producto) resolve({ id, precio: p.precio, tipo: p.tipo });
          else reject("ID no encontrado: " + id);
        }, 1000);
      }); return promesa;
};
```

De forma similar, **obtieneProductosPorTipo(tipo)** devolverá una promesa conteniendo un array de todos los productos que encajan con el tipo especificado: La promesa también se resolverá un segundo después de que se ejecute la función y se rechazará si se proporciona un tipo no válido:

```
const obtieneProductosPorTipo = (tipo) => {
        const promesa = new Promise(function(resolve,reject){
            let i = 0;
            let arrayTipos = [];
            let tiposPosibles = ['Electrónica','Libro','Ropa','Comida'];
            if(!tiposPosibles.includes(tipo))
                reject("Tipo no válido: " + tipo)
            else
                setTimeout(function(){
                    arrayTipos = catalogo.filter(p => p.tipo == tipo);
                    resolve(arrayTipos);
                },1000);
        }); return promesa;
};
```

Por último, **obtieneProductosPorPrecio** devolverá una promesa conteniendo un array de todos los productos que están dentro de la diferencia especificada sobre ese precio. La promesa se resolverá un segundo después de que la función se ejecute y se rechazará si se da un precio no válido.

```
const obtieneProductosPorPrecio = (precio, diferencia) => {
  const promesa = new Promise(function(resolve,reject){
    let i = 0;
    let arrayPrecios = [];

    if(!isFinite(precio)) reject("Precio no válido: " + precio)
    else setTimeout(function(){
       arrayPrecios = catalogo.filter(p => Math.abs(p.precio - precio) < diferencia);
      resolve(arrayPrecios);
     },1000);
    });

    return promesa;
};
```

8.3.2.2 Elementos visuales

En este apartado se verá cómo construir los elementos visuales de la aplicación; son los siguientes:

- Una sección "Buscar producto por Id" que incluya un campo de entrada y un botón etiquetado como "Buscar producto".
- Una sección "Producto examinado" que muestre id, precio y tipo de un producto.
- Una sección "Lista de productos similares que contenga una tabla que muestre una lista de productos similares.
- Una sección "Listar todos los productos" que contenga una tabla que muestre una lista de todos los productos.

Para ello, se añade el siguiente código en **catalogoProductos.html** el cual enlaza con los dos ficheros JavaScript requeridos por la aplicación:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <script src="lib.js"></script>
  </head>

  <body>
    <script src="catalogoProductos.js"></script>
  </body>
</html>
```

Justo antes de la etiqueta script, se añade el siguiente código en el body:

```
<div>
        <h3>Buscar producto por Id</h3>
        <input id="entrada"><button id="btnBuscar">Buscar producto</button>
</div>
<div>
        <h3>Producto examinado</h3>
        <p id="txtProducto">Id de producto: <br>Precio: <br>Tipo: </p>
        <h3>Lista de productos similares</h3>
        <table id="tblSimilar" width="300px" border="1" >
            <tr>
                <th>IdProducto</th><th>Tipo</th><th>Precio</th><th>Examinar</th>
            </tr>
        </table>
</div>
<div>
    <h3>Lista de todos los productos</h3>
        <table id="tblTodos" width="300px" border="1" >
           <tr>
               <th>IdProducto</th><th>Tipo</th><th>Precio</th><th>Examinar</th>
           </tr>
        </table>
</div>
```

El resultado final sería como el de la siguiente figura:

6. Interfaz de usuario de aplicación de productos

8.3.2.3 Lógica de catálogo de productos

En este apartado se incluye la lógica subyacente a la página de catálogo de productos, es decir, el contenido del archivo **catalogoProductos.js** que se incluye en el body.

La primera operación que se va a codificar es el listado de todos los productos. Se trata de poblar la tabla "Lista de todos los productos" usando la función correspondiente en lib.js. Para realizar esto, se necesita crear una función llamada **actualizaTabla(idTabla, arrayProductos)** que sirva de ayuda para rellenar dicha tabla.

También se necesita una función **creaEncabezadoTabla(idTabla)** encabezados dentro del método principal de actualización de la tabla.

Su código puede verse a continuación:

```
const creaEncabezadoTabla = (idTabla) => {

  let filaEncabezado = document.createElement("tr");

  let th1 = document.createElement("th");
  let th2 = document.createElement("th");
  let th3 = document.createElement("th");
  let th4 = document.createElement("th");

  th1.appendChild(document.createTextNode("IdProducto"));
  th2.appendChild(document.createTextNode("Tipo"));
  th3.appendChild(document.createTextNode("Precio"));
  th4.appendChild(document.createTextNode("Examinar"));

  filaEncabezado.appendChild(th1);
  filaEncabezado.appendChild(th2);
  filaEncabezado.appendChild(th3);
  filaEncabezado.appendChild(th4);

  document.getElementById(idTabla).appendChild(filaEncabezado);

};
```

El código a continuación se encarga de la actualización de la tabla. Para ello, referencia al elemento en el código HTML mediante el idTabla y añade dinámicamente las filas. Cada una de ellas incluirá el identificador de producto, tipo, precio y un botón Examinar con un manejador de evento que se verá posteriormente.

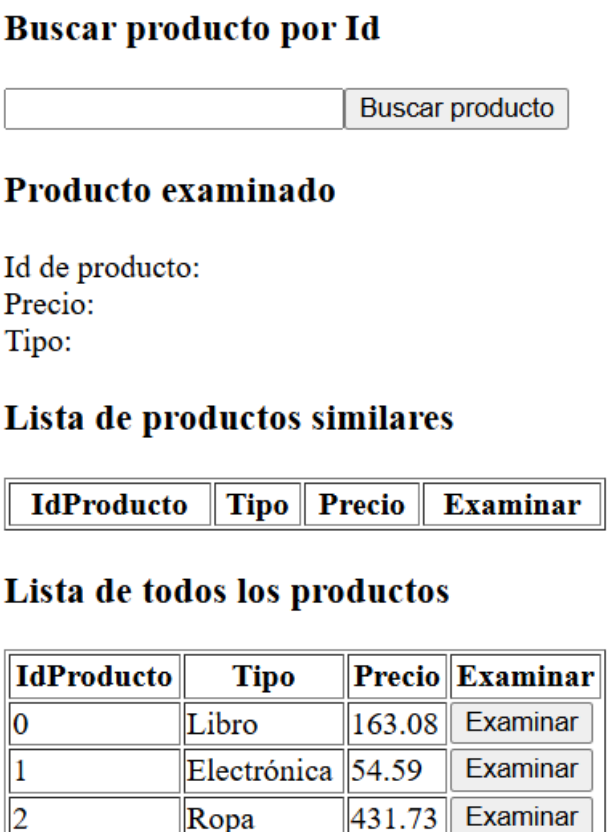
El código es el siguiente:

```
const actualizaTabla = (idTabla, arrayProductos) => {

  let bodyTabla = document.getElementById(idTabla);

  // Reinicia la tabla
  while (bodyTabla.hasChildNodes()) {
    bodyTabla.removeChild(bodyTabla.firstChild);
  }

  // Crea el encabezado de la tabla
  creaEncabezadoTabla(idtabla);

  // Publica las filas de la tabla
  arrayProductos.forEach((p) => {

    let tr = document.createElement("tr");

    let td1 = document.createElement("td");
    let td2 = document.createElement("td");
    let td3 = document.createElement("td");
    let td4 = document.createElement("button");

    td4.addEventListener("click", function () {});

    td1.appendChild(document.createTextNode(p.id));
    td2.appendChild(document.createTextNode(p.tipo));
    td3.appendChild(document.createTextNode(p.precio));
    td4.appendChild(document.createTextNode("Examinar"));

    tr.appendChild(td1);
    tr.appendChild(td2);
    tr.appendChild(td3);
    tr.appendChild(td4);

    bodyTabla.appendChild(tr);

  });
};
```

A partir de aquí ya se comienzan a usar funciones del fichero lib.js, concretamente el método **api.obtieneProductos()** con el que se obtiene una promesa que contiene un array de todos los productos en el catálogo. Después, mediante **actualizaTabla** se rellena la tabla con los productos. Para eso, se añade el siguiente código:

```
api.obtieneProductos().then(function (valor) {
  actualizaTabla("tblTodos", valor);
});
```

El resultado hasta el momento será similar al de la imagen:



7. Listado de productos

El siguiente paso es continuar editando el fichero **CatalogoProductos.js** para hacer lo siguiente:

- Rellenar la sección "Producto examinado" cuando se ha encontrado un id de producto o cuando se pulsa el botón Examinar.
- Rellenar la lista de productos similares con aquellos que sean parecidos al buscado/examinado.

Para hacer esto, primero se necesita crear una función llamada **actualizaExaminado(producto)** de forma que se rellene la sección producto examinado:

```
const actualizaExaminado = (producto) => {
  let salida = "Id de producto: " + producto.id;
  salida += "<br />Precio: " + producto.precio;
  salida += "<br> Tipo: " + producto.tipo;
  document.getElementById("txtProducto").innerHTML = salida;
}
```

Esta función edita el HTML en la sección "Producto examinado" y la rellena con los atributos del producto pasado.

La tabla de productos similares se rellena con productos del mismo tipo que el examinado y que están en un rango de precio de 50$ de diferencia. Por tanto, se pueden utilizar la funciones

api.obtieneProductosPorTipo(tipo) y **api.obtieneProductosPorPrecio(precio, diferencia)** y obtener la intersección de los dos arrays devueltos. Para eso se usará la siguiente función en **CatalogoProductos.js**:

```
const obtieneInterseccion = (arrA, arrB, idBuscada) =>
  arrA.filter((obj1) =>
    arrB.some((obj2) => obj1.id == obj2.id && obj1.id != idBuscada)
);
```

La función proporciona la intersección entre los dos arrays excluyendo idBusqueda, ya que obviamente, no debe aparecer en el array resultante. Ahora, se crea al fin un método de búsqueda, procesarBusqueda(idBusqueda) que irá a continuación del anterior en lib.js:

```
const procesarBusqueda = (idBusqueda) => {
  api.obtieneProductoPorId(idBusqueda).then(function(val){
      return
Promise.all([api.obtieneProductosPorPrecio(val.precio,50),api.obtieneProductosPorTipo
(val.tipo),val]);
  }).then(function(val){
      var arraySimilar = obtieneInterseccion(val[0],val[1],val[2].id);
      actualizaExaminado(val[2]);
      actualizaTabla('tblSimilar',arraySimilar);
  }).catch(function(val){
      alert(val);
  });
}
```

Esta función comienza usando la búsqueda de productos por id de la api para obtener una promesa conteniendo el producto buscado. Una vez que se resuelve esta promesa, entonces se encadena otra operación asíncrona devolviendo una llamada a Promise.all(). Este método procesa tres valores diferentes:

- Una promesa devuelta por api.obtieneProductosPorPrecio(val.precio,50)
- Una promesa devuelta por api.obtieneProductosPorTipo(val.tipo)
- El producto originalmente buscado representado por la variable val

El método Promise.all devuelve una promesa conteniendo un array que tiene una lista de productos con precios similares, otra de productos con tipos similares y el producto que se ha buscado originalmente. El método **obtieneIntersección** se usa para obtener la intersección entre los de precios similar y los de tipo similar omitiendo el producto del id dado. A continuación, se rellena la sección Producto examinado mediante **actualizaExaminado** con el producto original pasado como argumento mientras que la "Lista de productos similares" se rellena mediante **actualizaTabla** con el array obtenido de la intersección. Por último, si se busca un identificador no válido, el sistema mostrará un mensaje emergente.

Ahora que ya está definida la función **procesarBusqueda**, se puede usar en el código para actualizar la sección de producto examinado y la lista de productos similares.

Para ello, se añade el manejador de eventos en el botón de búsqueda:

```
document.getElementById("btnBuscar").addEventListener("click", function () {
  procesarBusqueda(document.getElementById("entrada").value);
});
```

A continuación, se modifica el manejador del evento click en la definición de **actualizaTabla** para que funcione el botón Examinar:

```
td4.addEventListener("click", function ()
{procesarBusqueda(this.parentNode.firstChild.innerHTML)});
```

Con esto ya se habría concluido el desarrollo, que, una vez probado, daría una salida similar a la de la siguiente imagen:

Buscar producto por Id

69 | Buscar producto

Producto examinado

Id de producto: 69
Precio: 158.42
Tipo: Ropa

Lista de productos similares

IdProducto	Tipo	Precio	Examinar
10	Ropa	142.16	Examinar
13	Ropa	151.33	Examinar
29	Ropa	168.80	Examinar
39	Ropa	178.82	Examinar
43	Ropa	177.56	Examinar
88	Ropa	123.98	Examinar

Lista de todos los productos

IdProducto	Tipo	Precio	Examinar
0	Ropa	210.75	Examinar
1	Ropa	3.23	Examinar
2	Ropa	92.89	Examinar
3	Libro	312.77	Examinar
4	Electrónica	365.43	Examinar
5	Comida	277.68	Examinar
6	Libro	120.10	Examinar
7	Libro	337.68	Examinar
8	Ropa	69.19	Examinar
9	Comida	474.26	Examinar

8. Listado de productos con promesas

A continuación, se incluye el código final de todos los elementos de esta aplicación. El código de lib.js:

```
(function (window) {
  const miBiblioteca = () => {

    // Las definiciones de función van aquí
    const creaObjetoAleatorio = () => {
      const arrayTipos = ["Electrónica", "Libro", "Ropa", "Comida"];
      const precio = (Math.random() * 500).toFixed(2);
      const tipo = arrayTipos[Math.floor(Math.random() * 4)];
      return { precio: precio, tipo: tipo };
    };

    const creaCatalogoAleatorio = (num) => {
      const catalogo = [];
      for (let i = 0; i < num; i++) {
        const obj = creaObjetoAleatorio();
        catalogo.push({ id: i, precio: obj.precio, tipo: obj.tipo });
      }
      return catalogo;
    };

    const obtieneProductos = () => {
      const promesa = new Promise(function (resolve, reject) {
        setTimeout(function () {
          resolve(catalogo);
        }, 1000);
      });
      return promesa;
    };

    const obtieneProductoPorId = (id) => {
      const promesa = new Promise(function (resolve, reject) {
        let i = 0;
        setTimeout(function () {
          console.log(catalogo);
          const producto = catalogo.find((p) => p.id == id);
          if (producto)
            resolve({ id, precio: producto.precio, tipo: producto.tipo });
          else reject("ID no encontrado: " + id);
        }, 1000);
      });
      return promesa;
    };
```

```
    const obtieneProductosPorTipo = (tipo) => {
      const promesa = new Promise(function (resolve, reject) {
        let i = 0;
        let arrayTipos = [];
        let tiposPosibles = ["Electrónica", "Libro", "Ropa", "Comida"];
        if (!tiposPosibles.includes(tipo)) reject("Tipo no válido: " + tipo);
        else
          setTimeout(function () {
            arrayTipos = catalogo.filter((p) => p.tipo == tipo);
            resolve(arrayTipos);
          }, 1000);
      });
      return promesa;
    };
    const obtieneProductosPorPrecio = (precio, diferencia) => {
      const promesa = new Promise(function (resolve, reject) {
        let i = 0;
        let arrayPrecios = [];
        if (!isFinite(precio)) reject("Precio no válido: " + precio);
        else
          setTimeout(function () {
            arrayPrecios = catalogo.filter(
              (p) => Math.abs(p.precio - precio) < diferencia
            );
            resolve(arrayPrecios);
          }, 1000);
      });
      return promesa;
    };

    // El código de ejecución va aquí
    const catalogo = creaCatalogoAleatorio(100);
    return {
      obtieneProductoPorId: obtieneProductoPorId,
      obtieneProductosPorPrecio: obtieneProductosPorPrecio,
      obtieneProductosPorTipo: obtieneProductosPorTipo,
      obtieneProductos: obtieneProductos,
    };
  };
  if (typeof window.api === "undefined") {
    window.api = miBiblioteca();
  }
})(window);
```

El de CatalogoProductos.html es el siguiente:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <script src="lib.js"></script>
  </head>

  <body>

    <div>
      <h3>Buscar producto por Id</h3>
      <input id="entrada" /><button id="btnBuscar">Buscar producto</button>
    </div>

    <div>
      <h3>Producto examinado</h3>
      <p id="txtProducto">Id de producto: <br />Precio: <br />Tipo:</p>
      <h3>Lista de productos similares</h3>
      <table id="tblSimilar" width="300px" border="1">
        <tr>
          <th>IdProducto</th>
          <th>Tipo</th>
          <th>Precio</th>
          <th>Examinar</th>
        </tr>
      </table>
    </div>

    <div>
      <h3>Lista de todos los productos</h3>
      <table id="tblTodos" width="300px" border="1">
        <tr>
          <th>IdProducto</th>
          <th>Tipo</th>
          <th>Precio</th>
          <th>Examinar</th>
        </tr>
      </table>
    </div>

    <script src="catalogoProductos.js"></script>

  </body>
</html>
```

Y, por último, el de CatalogoProductos.js:

```
const creaEncabezadoTabla = (idTabla) => {
  let filaEncabezado = document.createElement("tr");
  let th1 = document.createElement("th");
  let th2 = document.createElement("th");
  let th3 = document.createElement("th");
  let th4 = document.createElement("th");
  th1.appendChild(document.createTextNode("IdProducto"));
  th2.appendChild(document.createTextNode("Tipo"));
  th3.appendChild(document.createTextNode("Precio"));
  th4.appendChild(document.createTextNode("Examinar"));
  filaEncabezado.appendChild(th1);
  filaEncabezado.appendChild(th2);
  filaEncabezado.appendChild(th3);
  filaEncabezado.appendChild(th4);
  document.getElementById(idTabla).appendChild(filaEncabezado);
};
const actualizaTabla = (idTabla, arrayProductos) => {
  let bodyTabla = document.getElementById(idTabla);
  // Reinicia la tabla
  while (bodyTabla.hasChildNodes()) {
    bodyTabla.removeChild(bodyTabla.firstChild);
  }
  // Crea el encabezado de la tabla
  creaEncabezadoTabla(idTabla);
  // Publica las filas de la tabla
  arrayProductos.forEach((p) => {
    let tr = document.createElement("tr");
    let td1 = document.createElement("td");
    let td2 = document.createElement("td");
    let td3 = document.createElement("td");
    let td4 = document.createElement("button");
    td4.addEventListener("click", function () {
      procesarBusqueda(this.parentNode.firstChild.innerHTML);
    });
    td1.appendChild(document.createTextNode(p.id));
    td2.appendChild(document.createTextNode(p.tipo));
    td3.appendChild(document.createTextNode(p.precio));
    td4.appendChild(document.createTextNode("Examinar"));
    tr.appendChild(td1);
    tr.appendChild(td2);
    tr.appendChild(td3);
    tr.appendChild(td4);
    bodyTabla.appendChild(tr);
  });
};
```

```
api.obtieneProductos().then(function (valor) {
  actualizaTabla("tblTodos", valor);
});
const actualizaExaminado = (producto) => {
  let salida = "Id de producto: " + producto.id;
  salida += "<br />Precio: " + producto.precio;
  salida += "<br> Tipo: " + producto.tipo;
  document.getElementById("txtProducto").innerHTML = salida;
};
const obtieneInterseccion = (arrA, arrB, idBuscada) =>
  arrA.filter((obj1) =>
    arrB.some((obj2) => obj1.id == obj2.id && obj1.id != idBuscada)
);

const procesarBusqueda = (idBusqueda) => {
  api
    .obtieneProductoPorId(idBusqueda)
    .then(function (val) {
      return Promise.all([
        api.obtieneProductosPorPrecio(val.precio, 50),
        api.obtieneProductosPorTipo(val.tipo),
        val,
      ]);
    })
    .then(function (val) {
      var arraySimilar = obtieneInterseccion(val[0], val[1], val[2].id);
      actualizaExaminado(val[2]);
      actualizaTabla("tblSimilar", arraySimilar);
    })
    .catch(function (val) {
      alert(val);
    });
};
document.getElementById("btnBuscar").addEventListener("click", () => {
  procesarBusqueda(document.getElementById("entrada").value);
});
```

8.3.3 EJEMPLO COMPLETO: APLICACIÓN DE NOTAS

En este apartado se verá un ejemplo completo de programación de aplicación con comunicación asíncrona que soporta operaciones CRUD (Create, Read, Update y Delete).

Para ilustrar este ejemplo, se va a crear un script PHP que permita registrar una serie de notas a partir de su descripción e importancia, la cual puede tener un valor de 1 a 5.

El script puede recibir como nombres **notas.php** y almacenará cada uno de los registros en un archivo **notas.json**. Puede soportar los cuatro métodos HTTP siguientes:

- GET. Sirve para obtener un conjunto con todas las notas (sin parámetros) o una sola nota por su id (mediante http://ipscript/notas.php?id=identificador)
- POST. Permite dar de alta una nota enviando una cadena en formato JSON con el texto de la nota y su importancia.
- PUT. Actualiza una nota por su id. En este caso, el JSON debe incluir todos los campos, es decir, id, texto e importancia
- DELETE. Esta acción borra una nota por la id dada en formato JSON.

El archivo **notas.json** debe estar ubicado en el mismo que el script PHP. Se puede montar sobre un servidor web (Apache, nginx, etc.) o simplemente lanzando **php.exe -S localhost:8000** desde el directorio donde esté el fichero de script y el JSON.
El código completo de **notas.php** es el siguiente:

```php
<?php
header('Content-Type: application/json');
header('Access-Control-Allow-Origin: *');
header('Access-Control-Allow-Methods: GET, POST, PUT, DELETE, OPTIONS');
header('Access-Control-Allow-Headers: Content-Type');
if ($_SERVER['REQUEST_METHOD'] === 'OPTIONS') {
    header('HTTP/1.1 204 No Content');
    exit;
}
// Ruta del archivo JSON donde se almacenan las notas
$filePath = 'notas.json';
// Lee todas las notas existentes del fichero
function getAllNotas($filePath) {
    if (file_exists($filePath)) {
        $data = file_get_contents($filePath);

        // Se decodifica el JSON en un array asociativo
        $retValue = json_decode($data, true);

        // Por si acaso, se verifica que retValue es un array y que no está vacío
        if (is_array($retValue) && !empty($retValue)) {
            // Ahora se ordena por id
            usort($retValue, function ($a, $b) {
                return $a['id'] <=> $b['id']; // Operador de comparación espaciada
(<=>)
            });
        }
        // Se devuelve el array ordenado
        return $retValue;
    }
    // Si el archivo no existe o está vacío, se devuelve un array vacío
    return [];
}
```

```php
// Obtiene una nota por su ID.
 function getNotaById($id, $filePath) {
    $notas = getAllNotas($filePath);
    // Se usa array_filter con una expresión lambda para filtrar la nota por ID
    $notaFiltrada = array_filter($notas, function ($nota) use ($id) {
        return $nota['id'] == $id;
    });
    // Si se encontró la nota, se devuelve el primer (y único) elemento del array
resultante
    return !empty($notaFiltrada) ? array_values($notaFiltrada)[0] : null;
}



//Crea una nueva nota y la guarda en el archivo JSON.
 function createNota($texto, $importancia, $filePath) {
    $notas = getAllNotas($filePath);
    // Genera un nuevo ID único
    $id = isset($notas[count($notas) - 1]['id']) ? $notas[count($notas) - 1]['id'] +
1 : 1;
    // Crea la nueva nota
    $nuevaNota = [
        'id' => $id,
        'texto' => $texto,
        'importancia' => $importancia
    ];
    // Agrega la nueva nota al array de notas
    $notas[] = $nuevaNota;
    // Guardar las notas actualizadas en el archivo JSON
    file_put_contents($filePath, json_encode($notas, JSON_PRETTY_PRINT));
    return $nuevaNota;
}
// Actualiza una nota existente por su ID.
 function updateNotaById($id, $texto, $importancia, $filePath) {
    $notas = getAllNotas($filePath);
   // Se usa array_map con una expresión lambda para actualizar la nota si coincide
con el ID
   $notasActualizadas = array_map(function ($nota) use ($id, $texto, $importancia) {
    if ($nota['id'] == $id) {
        // Actualizamos los campos de la nota
        return [
            'id' => $id,
            'texto' => $texto,
            'importancia' => $importancia
        ];
    }
    return $nota; // Si no coincide el ID, se deja la nota sin cambios
}, $notas);
```

```
    // Guarda las notas actualizadas en el archivo JSON
    file_put_contents($filePath, json_encode($notasActualizadas, JSON_PRETTY_PRINT));
    return getNotaById($id, $filePath);
}
// Elimina una nota por su ID.
 function deleteNotaById($id, $filePath) {
    $notas = getAllNotas($filePath);
    // Deja todas las notas excepto la del ID dado
    $notasFiltradas = array_filter($notas, function ($nota) use ($id) {
        return $nota['id'] != $id;
    });



    // Guarda las notas actualizadas en el archivo JSON
    file_put_contents($filePath, json_encode(array_values($notasFiltradas),
JSON_PRETTY_PRINT));
    return ['message' => "Nota con ID $id eliminada correctamente."];
}
// Procesa las solicitudes según el método HTTP
switch ($_SERVER['REQUEST_METHOD']) {
    case 'GET':
        if (isset($_GET['id'])) {
            $id = intval($_GET['id']);
            $nota = getNotaById($id, $filePath);
            echo json_encode($nota ?: ['error' => "No se encontró ninguna nota con ID
$id."]);
        } else {
            $notas = getAllNotas($filePath);
            echo json_encode($notas);
        }
        break;
    case 'POST':
        $input = json_decode(file_get_contents('php://input'), true);
        if (isset($input['texto']) && isset($input['importancia'])) {
            $nuevaNota = createNota($input['texto'], $input['importancia'],
$filePath);
            http_response_code(201); // Código de creación exitosa
            echo json_encode($nuevaNota);
        } else {
            http_response_code(400); // Código de error: solicitud incorrecta
            echo json_encode(['error' => 'Datos incompletos para crear la nota.']);
        }
        break;
```

```
    case 'PUT':
        $input = json_decode(file_get_contents('php://input'), true);
        if (isset($input['id']) && isset($input['texto']) &&
isset($input['importancia'])) {
            $actualizada = updateNotaById($input['id'], $input['texto'],
$input['importancia'], $filePath);
            echo json_encode($actualizada);
        } else {
            http_response_code(400); // Código de error: solicitud incorrecta
            echo json_encode(['error' => 'Datos incompletos para actualizar la
nota.']);
        }
        break;
    case 'DELETE':
        $input = json_decode(file_get_contents('php://input'), true);
        if (isset($input['id'])) {
            $resultado = deleteNotaById($input['id'], $filePath);
            echo json_encode($resultado);
        } else {
            http_response_code(400); // Código de error: solicitud incorrecta
            echo json_encode(['error' => 'ID de nota no proporcionado para
eliminar.']);
        }
        break;

    default:
        http_response_code(405); // Método no permitido
        echo json_encode(['error' => 'Método no permitido.']);
}
?>
```

La aplicación cliente se realizará en React para ilustrar el uso de componentes y funciones asíncronas mediante esta biblioteca.

8.3.3.1 Biblioteca con las operaciones de servidor

Antes de comenzar el diseño de la interfaz de usuario en React, lo primero que hay que hacer es aislar las operaciones CRUD que se realizarán contra el servidor. Para ello, se creará un archivo JavaScript que se puede llamar **notasServer.js** ya que contendrá las funciones de acceso al servidor para el manejo de notas. En primer lugar, se especifica la URL del servidor al que irán dirigidas las llamadas, es decir, la que contendrá la API de gestión de las notas:

```
const url = "http://localhost:8000/notas.php";
```

Tanto si se está trabajando con servidor ad-hoc (mediante php –S localhost:8000) como con otro externo, es muy importante que esté en local cuando se esté en desarrollo ya que es donde se ubica la aplicación React. Esto evita problemas de CORS, es decir, el bloqueo de acceso a URLs externas. En cualquier caso, el script PHP ya está preparado para lidiar con esto.

El primer método permitirá obtener todas las notas del servidor:

```
// Obtención de todas las notas
export const getAllNotas = async () => {

  const mensajeError = "Error al obtener todas las notas";

  try {
    const respuesta = await fetch(url);
    if (!respuesta.ok) throw new Error (mensajeError);
    return await respuesta.json();
  }
  catch (error)
  {
    console.error(mensajeError, error);
    return [];
  }
};
```

Como puede verse, usa async / await combinado con fetch, tal como se ha visto en otros ejemplos. La petición de todas las notas se hace a través de una petición get sin argumentos a la URL base. Cuando termine la petición, devolverá una cadena JSON con todas las notas.

A partir de aquí, se podría obtener cualquier nota por identificador con un simple filtrado sobre este JSON, pero, ya que en el servidor se puede realizar una petición GET para obtener una sola, se creará un método para dar soporte a esta operación, principalmente a la hora de leer cada una de ellas. El método para obtener una nota por identificador es el siguiente:

```
// Obtención de una nota por ID
export const getNotaById = async (idNota) => {
  let mensajeError = `Error al obtener la nota con id ${idNota}`;
  try {
    const respuesta = await fetch(`${url}?id=${idNota}`);
    if (!respuesta.ok) throw new Error(mensajeError);
    return await respuesta.json();
  } catch (error) {
    console.error(mensajeError, error);
    return null;
  }
};
```

Bien, hasta el momento ya se tienen los dos métodos de lectura. Ahora toca crear métodos que permitan añadir registros, modificarlos y borrarlos. El primero de ellos estará encaminado a dar de alta una nota. Esta operación se hace mandando una petición POST e incluyendo en su body el JSON de la nota a dar de alta. Hay que tener en cuenta que, en este caso, el JSON de la nota no debe contener el id, ya que este se obtiene automáticamente del servidor.

Como puede verse en el código, se utiliza un initObject para completar la petición.

```
// Añadir nueva nota
export const addNota = async (nuevaNota) => {
  const initObject = {
    method: "POST",
    headers: { "Content-Type": "application/json" },
    mode: "cors",
    body: JSON.stringify(nuevaNota),
  };
  const mensajeError = "Error al añadir la nota";
  try {
    const respuesta = await fetch(url, initObject);
    if (!respuesta.ok) throw new Error(mensajeError);
    return await respuesta.json();
  } catch (error) {
    console.error(mensajeError, error);
    return null;
  }
};
```

Por ejemplo, una nota válida para dar de alta tendría el siguiente formato JSON:

```
{ "texto": "Nueva nota", "importancia": 3}
```

El actualizado de una nota se hace mediante el método PUT de HTTP.

```
// Actualizar una nota existente
export const updateNota = async (actNota) => {
  const initObject = {
    method: "PUT",
    headers: { "Content-Type": "application/json" },
    mode: "cors",
    body: JSON.stringify(actNota),
  };
  const mensajeError = "Error al actualizar la nota";
  try {
    const respuesta = await fetch(url, initObject);
    if (!respuesta.ok) throw new Error(mensajeError);
    return await respuesta.json();
  } catch (error) {
    console.error(mensajeError, error);
    return null;
  }
};
```

La nota a actualizar en este caso debe ser un JSON que contenga los tres campos: id, texto e importancia. En este caso se actualizaría la nota número 5:

```
{ "id": 5, "texto": "Esta es una nota actualizada", "importancia": 5}
```

Por último, el borrado se haría utilizando DELETE:

```
export const deleteNota = async (idNota) => {

  const initObject = {
    method: "DELETE",
    headers: { "Content-Type": "application/json" },
    mode: "cors",
    body: JSON.stringify({id: idNota}),
  };

  const mensajeError = "Error al eliminar la nota";

  try
  {
    const respuesta = await fetch(url, initObject);
    if (!respuesta.ok) throw new Error(mensajeError);
    return true;
  }
  catch (error)
  {
    console.error(mensajeError, error);
    return false;
  }
};
```

A diferencia del método de obtención, dado que la llamada a este método en el servidor se hace mediante DELETE y no GET, ese incluye el identificador a borrar en el cuerpo de la petición. Por tanto, este método debería ser llamado con un JSON como este:

```
{ "id": 7 }
```

Lo que provocaría la eliminación de la nota con identificador 7.

8.3.3.2 Componentes React

En este apartado se van a detallar los componentes necesarios para hacer funcionar este sistema. La arquitectura elegida se va a basar en dos componentes:

- ListadoNotas: Su misión es mostrar una tabla con todas las notas almacenadas en el servidor. Para cada una de ellas, el sistema añadirá dos botones: uno para actualizar la nota y otro para eliminarla. Por otra parte, contendrá un botón que permita añadir una nueva nota. Este componente contendrá a su vez el que se detalla a continuación.
- FormNota: Cuando se realice una operación de agregado de nota o actualización, se mostrará un pequeño formulario de edición con los datos de la nota.

La aplicación utilizará estados y propiedades para la gestión de la información que se manda entre componentes añadiendo métodos que implican a ambos.

Antes de ver el código JavaScript del componente ListadoNotas, se incluyen los estilos que se aplican a dicho componente en el archivo ListadoNotas.css:

```css
/* Estilo base de la tabla */

.tabla { width: 100%;
  border-collapse: collapse;
  border: 1px solid #ddd;
  margin: 20px 0;
}

.tabla th {
  background-color: #f4f4f4;
  color: #333;
  font-weight: bold;
  padding: 10px;
  text-align: center;
  border: 1px solid #ddd;
}

.tabla fila { border-bottom: 1px solid #ddd;}

.tabla td { padding: 10px;  border: 1px solid #ddd; }

.celda-id { text-align: right;}

.celda-texto {
  text-align: left;
}

.celda-importancia {
  text-align: center;
}

.fila:hover {  background-color: #f9f9f9;}
```

El código del componente ListadoNotas.js se muestra a continuación:

```
import "./ListadoNotas.css";
import { useState, useEffect } from "react";
import { getAllNotas, deleteNota, updateNota, addNota } from "../notasServer";
import FormNota from "./FormNota";
const ListadoNotas = () => {
  const [notas, setNotas] = useState([]);
  const [notaActual, setNotaActual] = useState(null);
  const [formVisible, setFormVisible] = useState(false);
  const loadNotas = async () => {
    try {
      const notasServer = await getAllNotas();
      setNotas(notasServer);
    } catch (error) {
      console.error("Error al cargar las notas:", error);
    }
  };
  const showForm = (nota) => {
    setNotaActual(nota);
    setFormVisible(true);
  };
  const updateNotaEvt = (nota) => showForm(nota);
  const deleteNotaEvt = async (idNota) => {
    let mensajeError = `Se ha producido un error al borrar la nota con identificador
${idNota}`;
    try {
      const exito = await deleteNota(idNota);
      if (!exito) alert(mensajeError);
      else {
        alert(`La nota con identificador ${idNota} se ha borrado correctamente`);
        loadNotas();
      }
    } catch (error) {
      alert(mensajeError, error);
    }
  };
  const saveNotaEvt = async (nota) => {
    try {
      // Si la nota tiene id, se trata de una actualización, si no, es nueva
      if (nota.id) {
        await updateNota(nota);
        alert(
          `Se ha modificado la nota con identificador ${nota.id} correctamente`
        );
      } else {
        const nuevaNota = await addNota(nota);
        alert(`La nota se ha añadido correctamente con id ${nuevaNota.id}`);
      }
```

```
      // Se recargan las notas y se oculta el form
      loadNotas();
      setFormVisible(false);
    } catch (error) {
      alert("Se ha producido un error al guardar la nota: ", error);
    }
  };
  useEffect(() => { loadNotas(); }, []);
  return (
    <>
      <button onClick={() => showForm()}>
        Añadir Nota
      </button>
      <table className="tabla">
        <thead>
          <tr>
            <th>ID</th>
            <th>Texto</th>
            <th>Importancia</th>
          </tr>
        </thead>
        <tbody>
          {notas?.map((nota) => (
            <tr key={nota.id} className="fila">
              <td className="celda-id">{nota.id}</td>
              <td className="celda-texto">{nota.texto}</td>
              <td className="celda-importancia">{nota.importancia}</td>
              <td>
                <button onClick={() => updateNotaEvt(nota)}>Actualizar</button>
                <button onClick={() => deleteNotaEvt(nota.id)}>Borrar</button>
              </td>
            </tr>
          ))}
        </tbody>
      </table>

      {formVisible && (
        <FormNota
          nota={notaActual}
          onSave={saveNotaEvt}
          onCancel={() => setFormVisible(false)}
        />
      )}
    </>
  );
};
export default ListadoNotas;
```

Como puede verse, el componente tiene muchas cuestiones que explicar. En primer lugar, se observa que consta de tres variables de estado:

- notas. Contiene la colección completa de notas obtenidas del servidor
- notaActual. En esta propiedad se almacena la nota actual que se puede estar editando o añadiendo en el formulario de edición
- formVisible. Flag que controla la visibilidad o no del formulario de notas

El siguiente método **loadNotas()** obtiene las notas actualizadas del servidor y las ubica en la variable de estado produciéndose la actualización de la tabla que las contiene.

El método **showForm(nota)** modifica la variable de estado de la nota actual y muestra el formulario de edición.

A continuación, se añaden algunos de los manejadores de eventos, tanto para actualización como para borrado. En lo que respecta al primero, se crea una función que en realidad simplemente muestra el formulario con una nota dada. El manejador es exactamente el mismo que se usa para añadir una nueva nota, solo que en caso de que la nota sea nueva, esta variable no estará definida y se hará al pulsar el botón de guardado en el formulario. El método de borrado es directo, ya que no depende del formulario. Su funcionamiento es asíncrono, lanzando el método de servidor **deleteNota** con el identificador de la nota a la que está referido. El mapeo de notas a eventos y a cada una de las filas de la tabla se hace en el renderizado, por lo que el sistema ya sabe que cuando se hace clic sobre el botón de la nota con identificador 3, por ejemplo, debe lanzar el manejador **deleteNotaEvt(3)**.

El manejador **saveNotaEvt** no se va a usar en **ListadoNotas**, sino en el componente formulario ya que será el que gestione la creación o actualización de una nota. ¿Y por qué se ubica aquí y no en el componente hijo? Básicamente porque hay algo que debe hacerse en el padre y es la actualización del listado de notas cuando se produzca alguna operación de creación / actualización / borrado. Por tanto, se define el cuerpo de este manejador en este componente y se le pasa como propiedad al componente hijo. En este evento es interesante ver que si se le pasa una nota con id, la va a modificar y si no lo tiene, la creará en el servidor. Los últimos pasos son la recarga de los datos y la ocultación del formulario.

La parte de lógica pura termina con el hook **useEffect** que se usa para ejecutar código en el renderizado del componente. Básicamente, lo único que hace es cargar la tabla con los datos del servidor.

En cuanto a la especificación propia del componente, el primer elemento es el botón de añadir, que se ubica en la parte superior para que no se mueva con cada cambio de tamaño de la tabla. Este botón lo único que hace es mostrar el formulario, pero sin ninguna nota, con lo que implícitamente ya se está dando por hecho que tiene que crear una nueva. Mediante **map** se rellenan las filas de la tabla añadiendo los botones ya citados y se cierra con el control formulario que se verá a continuación.

El código de FormNota.js es el siguiente:

```
import { useEffect, useState } from "react";
const FormNota = ({ nota, onSave, onCancel }) => {
  const [texto, setTexto] = useState("");
  const [importancia, setImportancia] = useState(3);
  const envioEvt = (e) => {
    e.preventDefault();
    // notaGuardar va a basarse en la nota existente si es actualización
    // o a crear una nueva si es añadido
    const notaGuardar = {
      ...(nota || {}), // Si la nota ya existe, se mantiene su id
      texto,
      importancia: parseInt(importancia),
    };
    // Ahora se llama al evento onSave que está en el control padre
    onSave(notaGuardar);
  };
  const cancelarEvt = () => { if (onCancel) onCancel(); };
  useEffect(() => {
    // Si hay nota, se cargan sus valores
    if (nota) {
      setTexto(nota.texto);
      setImportancia(nota.importancia);
    }
  }, [nota]);
  return (
    <form onSubmit={envioEvt}>
      <h2>{nota ? "Actualizar" : "Añadir"}</h2>
      <label>
        Texto:
        <input type="text" value={texto} onChange={(e) => setTexto(e.target.value)}
          required />
      </label>
      <label>
        Importancia (1-5):
        <input type="number" min="1" max="5" value={importancia}
          onChange={(e) => setImportancia(e.target.value)} required />
      </label>
      <button type="submit">{nota ? "Guardar" : "Añadir"}</button>
      <button type="button" onClick={cancelarEvt}>
        Cancelar
      </button>
    </form>
  );
};
export default FormNota;
```

Como puede verse, el control recoge tres propiedades: la nota base del formulario y las funciones **onSave** y **onCancel**. Además, contiene dos variables de estado, una por cada campo del formulario.

El manejador **envioEvt** se lanza al pulsar el botón guardar. Básicamente, lo que hace es construir una nota con las variables de estado y mantiene el id si es que nota está definida para cuando se trate de una actualización. Termina con la llamada a ese **onSave** delegado desde el control padre para completar el guardado en el servidor y actualizar la tabla. Se hace lo mismo con el evento para el botón cancelar, aunque en este caso la lógica com, es mucho más simple.

En el **useEffect** se cargan las variables de estado si se ha pasado una nota como propiedad.

En cuanto a la parte propia del renderizado del componente, se usa el concepto de formulario controlado actualizando en tiempo real las propiedades mediante el método **onChange** de los elementos INPUT.

Por último, solo quedaría incrustar el componente ListadoNotas en el App.js:

```
import ListadoNotas from "./ListadoNotas";
function App() {
  return (
    <div>
      <h1>GESTIÓN DE NOTAS</h1>
      <ListadoNotas />
    </div>
  );
}
```

ÍNDICE DE FIGURAS

BIBLIOGRAFÍA – WEBGRAFÍA

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

Microsoft (2018). *Asynchronous Programming with Javascript.* https://learning.edx.org/course/course-v1:Microsoft+DEV234x+3T2018/

John.S, T. (2023) *Promises, Async/Await, and their Role in Modern Web Development – Asynchronous Programming in JavaScript.* https://medium.com/@stheodorejohn/promises-async-await-and-their-role-in-modern-web-development-asynchronous-programming-in-6c6d01658e25

Bilal (2024). *How to Master JavaScript's Async/Await Like a Pro.* https://medium.com/@Bilal.se/how-to-master-javascripts-async-await-like-a-pro-b20338eca0f8