

Universidad Tecnológica Nacional
Facultad Regional Rosario
Tecnicatura Universitaria en Programación



Programación III

Unidad 4.1

Efectos secundarios

Efectos secundarios o paralelos	3
¿Qué son los efectos secundarios?	3
useEffect y sus dependencias	4
Conexión de lado cliente con lado servidor	4
Problemas de CORS	5
Sincronizando estado con la respuesta del servidor	6
Agregar un libro: método POST	7
Validaciones	8
Editar un libro	11
Ejercicio de clase - eliminar un libro	17

Efectos secundarios o paralelos

¿Qué son los efectos secundarios?

El objetivo principal de React se divide en dos partes: el **renderizado de la interfaz de usuario** y **reaccionar ante los *inputs* del usuario**. Dentro de esos dos objetivos se encuentran sub-objetivos tales como:

- Evaluar y renderizar JSX.
- Manejar el *state* y las *props*.
- Reaccionar a eventos e *inputs* que genera el usuario.
- Re-evaluar componentes ante cambios de *props* y *state*.

Todo esto lo cubre la librería React mediante las herramientas y *features* que ya vienen consigo misma.

¿Qué son entonces los efectos, o los efectos secundarios? Bueno, **todo lo demás**.

Son tareas que suceden por fuera del ciclo de vida normal de los componentes, ya que pueden llegar a bloquear o a retrasar el renderizado de los mismos. Los efectos se usan en general para:

- Mandar pedidos al servidor mediante el protocolo HTTP.
- Guardar data en el Data Storage del navegador.
- Poner y manejar temporizadores.

Para manejar los efectos secundarios, React nos provee con otro *hook* en su librería: ***useEffect***.

```
useEffect( () => {}, [ dependencias ] );
```

Analicemos su estructura:

- Posee dos parámetros:
 - El primero es una función (una *callback*, como una *arrow function*) que se ejecutará **después** de cada evaluación del componente **solo si las dependencias especificadas cambiaron**.
 - El segundo parámetro es un arreglo donde se escriben las dependencias que ***useEffect*** debe escuchar.

useEffect y sus dependencias

El arreglo de dependencias tiene 3 valores posibles, y dependiendo de ese valor, la *callback* que se encuentra en el primer parámetro del *useEffect* se ejecutará en cierto momento del ciclo de vida del componente.

Los 3 valores son:

- ***null***: si no ponemos el segundo parámetro, el *useEffect* se va a ejecutar en **cada renderizado del componente**. Esto hace que el *hook* no sea efectivo en su responsabilidad, por ende **no se utiliza en ningún caso de uso**. Solo lo mencionamos como posibilidad.
- **Arreglo vacío** (`[]`): es el más utilizado, ya que es el más confiable. Se ejecuta **una sola vez, después del primer renderizado del componente**. Se usa en general para hacer llamadas necesarias al servidor y traer datos una vez que cargue la pantalla deseada (veremos un sencillo ejemplo más adelante)
- **Arreglo con diferentes dependencias** (`[dep1, dep2, ...]`): va a forzar la ejecución del *useEffect* **si y sólo si ha cambiado de valor alguna de las dependencias**. Esta forma de escribir el *useEffect* tiende a generar errores graves de performance (re-renderizados infinitos, llamadas extras al servidor no deseadas). Tal es así, que React escribió una página entera donde cita todos los ejemplos en donde el *useEffect* no es recomendable de usar [You might not need an effect](#)

Conexión de lado cliente con lado servidor

Ya en el apunte 3.3 creamos un servidor funcional para nuestro lado cliente. Solo nos resta realizar la unión de ambas partes. Lo intentaremos primero con el GET de todos los libros de la base de datos.

Para ello, utilizaremos la función *fetch*, nativa de javascript:

```

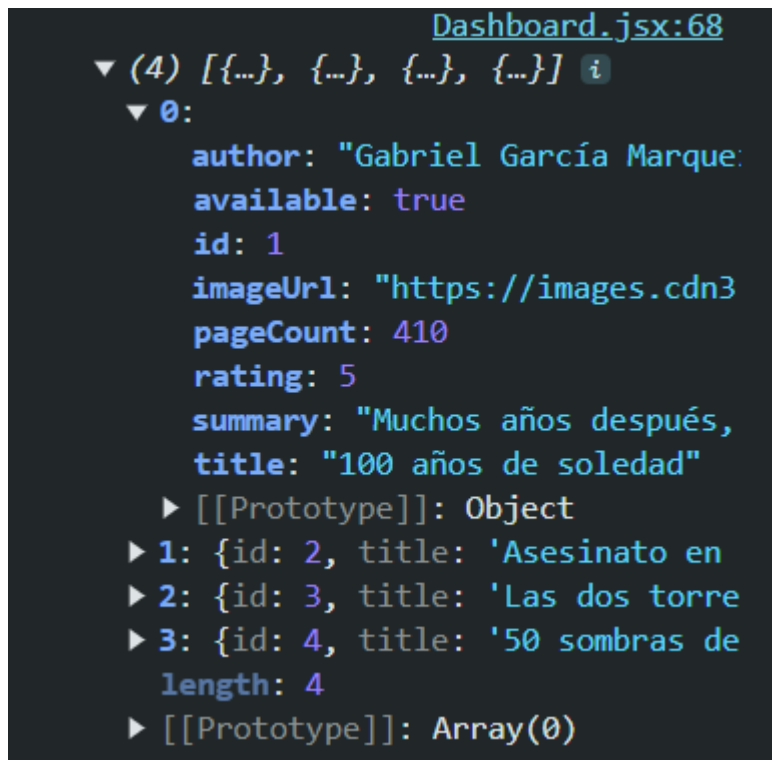
65     useEffect(() => {
66         fetch("http://localhost:3000/books")
67             .then(res => res.json())
68             .then(data => console.log(data))
69             .catch(err => console.log(err));
70     }, [])
71 
```

Analicemos la función anterior:

- *fetch* es una función asíncrona de javascript que me permite realizar *requests* al servidor. Posee dos parámetros:
 - La URL / *endpoint* donde queremos realizar el pedido.

- El método a utilizar, en este caso es GET (no es necesario especificarlo, ya que es el método *default* de *fetch*)
- Un objeto javascript con los headers requeridos de la *request* (en ellos puede ir el token, el tipo de archivo que esperamos recibir / mandar, etc). Veremos más sobre configuración en el método POST.
- El cuerpo de la *request*, denominado *body*. El GET, al no necesitar información adicional más que la ruta (en este caso), no tiene cuerpo la *request*.

Luego, comprobaremos que la *response* sea correcta.



```

Dashboard.jsx:68
▼ (4) [{...}, {...}, {...}, {...}] ⓘ
  ▼ 0:
    author: "Gabriel García Marque:
    available: true
    id: 1
    imageUrl: "https://images.cdn3
    pageCount: 410
    rating: 5
    summary: "Muchos años después,
    title: "100 años de soledad"
    ► [[Prototype]]: Object
  ► 1: {id: 2, title: 'Asesinato en
  ► 2: {id: 3, title: 'Las dos torre
  ► 3: {id: 4, title: '50 sombras de
    length: 4
  ► [[Prototype]]: Array(0)
  
```

Problemas de CORS

Si la *response* no se loguea como la imagen anterior, puede que poseamos un problema de CORS (incluso, sería correcto tener este error) ¿A qué nos referimos con un error de CORS?

CORS (*Cross-Origin Resource Sharing*) es un sistema que consiste en transmitir HTTP headers, que determina si los navegadores bloquean el acceso del código JavaScript frontend a las respuestas de peticiones de origen cruzado. En nuestro caso, las peticiones de nuestro *localhost:5173* están siendo bloqueadas por el servidor, al ser un origen no permitido (el origen del código de servidor, obviamente si está permitido ejecutarse).

Para ello, en nuestro *index.js* podemos especificar que acepte cualquier origen cruzado de la siguiente manera:

```
14 try {
15   app.use(express.json())
16   app.use((req, res, next) => {
17     res.header("Access-Control-Allow-Origin", "*");
18     res.header("Access-Control-Allow-Headers", "*");
19     res.header("Access-Control-Allow-Methods", "GET, POST, PUT, DELETE");
20     next();
21   });
22   app.listen(PORT);
23   app.use(bookRoutes);
24   app.use(authRoutes);
25 }
```

La función *next* se utiliza para pasar al siguiente middleware, sin terminar la ejecución del ciclo *request-response* (manteniendo las configuraciones realizadas sobre la *request* y/o la *response*)

Es necesario decir que en producción, **este código es altamente riesgoso**. En un ambiente real (no de desarrollo), vamos a desear restringir lo más posible el número de URLs que puedan comunicarse con el servidor.

Sincronizando estado con la respuesta del servidor

Finalmente, debemos actualizar nuestro lado cliente con la información recibida por el servidor (la lista de libros).

Sencillamente, realizamos un *setState* con dicha información.

```
65 useEffect(() => {
66   fetch("http://localhost:3000/books")
67     .then(res => res.json())
68     .then(data => setBookList([...data]))
69     .catch(err => console.log(err));
70 }, [])
```

Nota: no es necesario realizar el seteo de estado con el *spread operator*, ya que sería lo mismo si hacemos *setBookList(data)*. Es solo una forma más prolija de hacerlo, donde informamos al otro desarrollador "Lo que devuelve este método es un arreglo"

Agregar un libro: método POST

Fijémonos en el componente donde agregamos un libro, Dashboard.

```

64
65   useEffect(() => {
66     fetch("http://localhost:3000/books")
67       .then(res => res.json())
68       .then(data => console.log(data))
69       .catch(err => console.log(err));
70   }, []);
71
72   const handleBookAdded = (enteredBook) => {
73     const bookData = {
74       ...enteredBook,
75       id: Math.random()
76     }
77
78     setBookList(prevBookList => [bookData, ...prevBookList])
79   }
80
81   const handleDeleteBook = (id) => {
82     setBookList(prevBookList => prevBookList.filter(book => book.id !== id));
83   }
84
85   const handleNavigateAddBook = () => {
86     navigate("add-book");
87   }
88

```

¿Dónde creen ustedes que deberíamos hacer la llamada al servidor? ¿Nos toca crear un nuevo *useEffect*?

Este es el clásico ejemplo donde **no debemos utilizar *useEffect***. Si el efecto queremos ejecutarlo atado a un evento, **debe ejecutarse dentro de la función manejadora de ese evento**.

La llamada entonces la haremos dentro de *handleBookAdded*.

```

72   const handleBookAdded = (enteredBook) => {
73     fetch("http://localhost:3000/books", {
74       headers: {
75         "Content-type": "application/json"
76       },
77       method: "POST",
78       body: JSON.stringify(enteredBook)
79     })
80       .then(res => res.json())
81       .then(data => {
82         setBookList(prevBookList => [data, ...prevBookList])
83       })
84       .catch(err => console.log(err))
85   }
86

```

Analicemos un poco cómo cambió la función

- Agregamos el segundo parámetro luego de la URL, que es el parámetro de configuración de la *request*. Allí seteamos
 - *headers*: solamente para especificar que el contenido que enviamos es del tipo JSON.
 - *method*: ya que el método por default es GET, debemos especificar también que este será un método POST.
 - *body*: el cuerpo de la *request*, con la data necesaria para la creación del nuevo libro. En este caso, realizamos un *stringify* para poder enviar el JSON en forma de string, que es lo que nos pide fetch API.
- En la resolución de la promesa final, seteamos el estado de manera similar, cambiando el libro por parámetro por la data que devuelve el pedido al servidor.

Validaciones

Recordemos que el título y el autor eran campos obligatorios para la creación de una nueva entidad Libro. Si clickeamos el botón “Agregar libro” sin el título o el autor, ¿Qué sucede?

El servidor nos da un error (400) y no nos agrega el libro en la base de datos, pero el usuario no se entera de esto. Solo ve como el formulario se limpia, y quizás piense “No se que pasó, pero supongo que el libro se agregó”. O peor aún, puede pensar que no se agregó y seguir intentando por un rato largo.

Mantener informado al usuario es parte de las [10 heurísticas de diseño de Jakob Nielsen](#), específicamente la primera:

Visibility of system status: keep users in the loop by providing timely and relevant feedback. This helps build trust and enable users to make informed decisions about their next steps

Podemos utilizar para los mensajes al usuario, una librería muy útil llamada [react-toastify](#)

Primero la instalamos mediante:

```
npm i --save react-toastify
```

Luego, para que podamos emitir mensajes dentro de toda la app, la librería nos recomienda que pongamos un contenedor que funciona a modo de *provider* (veremos bien que son estos *providers* en el apunte de Contexto).

¡Y listo! Ya podemos armar nuestros mensajes *toast* para cada situación:


```
74 | if (!enteredBook.title || !enteredBook.author) {  
75 |     toast.error('El autor y/o título son requeridos', {  
76 |         position: "top-right",  
77 |         autoClose: 5000,  
78 |         hideProgressBar: false,  
79 |         closeOnClick: false,  
80 |         pauseOnHover: true,  
81 |         draggable: true,  
82 |         progress: undefined,  
83 |         theme: "light",  
84 |         transition: Bounce,  
85 |     });  
86 |     return;  
87 | }
```

El autor y/o título son requeridos

Book champions app

¡Quiero leer libros!

Título

Ingresa título

Autor

Ingresa autor

Puntuación

Ingresa cantidad de estrellas

Cantidad de páginas

Ingresa cantidad de páginas

URL de imagen

Ingresa url de imagen

¿Disponible?

☐

Volver

Agregar lectura

Como los mensajes de error / información / advertencia, poseen una lógica muy similar entre sí, armemos un archivo js común denominado *notifications.js*, que solo reciba por parámetros el mensaje a mostrar y configuraciones diferentes, en caso de ser necesario.

```

1  import { Bounce, toast } from "react-toastify";
2
3  const defaultNotificationConfig = {
4    position: "top-right",
5    autoClose: 5000,
6    hideProgressBar: false,
7    closeOnClick: false,
8    pauseOnHover: true,
9    draggable: true,
10   progress: undefined,
11   theme: "light",
12   transition: Bounce,
13 }
14
15 export const errorToast = (message, config) => {
16   return toast.error(message, {
17     ...defaultNotificationConfig,
18     ...config
19   })
20 }

```

A su vez, sería correcto agregar un mensaje de éxito cuando el libro se agrega correctamente.

```

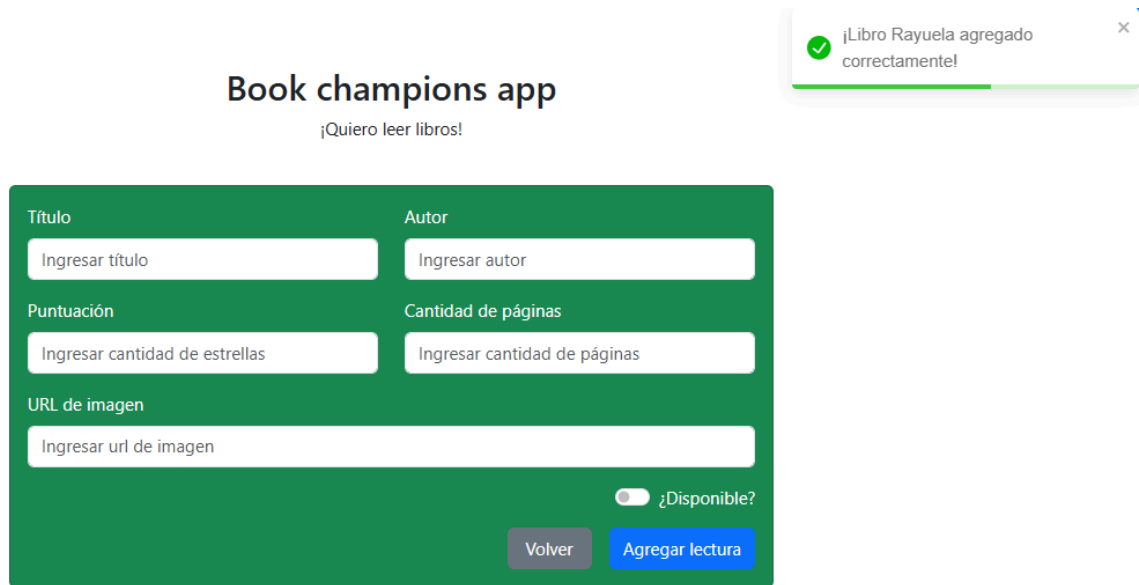
22 export const successToast = (message, config) => {
23   return toast.success(message, {
24     ...defaultNotificationConfig,
25     ...config
26   })
27 }

```

```

.then(data => {
  setBookList(prevBookList => [data, ...prevBookList]);
  successToast(`¡Libro ${data.title} agregado correctamente!`)
})

```



The screenshot displays the 'Book champions app' interface. At the top, a green header contains the app name and the text '¡Quiero leer libros!'. Below this is a form with a dark green background and white text. The form has five input fields: 'Título' (Title), 'Autor' (Author), 'Puntuación' (Rating), 'Cantidad de páginas' (Number of pages), and 'URL de imagen' (Image URL). Each field has a placeholder text: 'Ingresar título', 'Ingresar autor', 'Ingresar cantidad de estrellas', 'Ingresar cantidad de páginas', and 'Ingresar url de imagen' respectively. To the right of the 'URL de imagen' field is a toggle switch labeled '¿Disponible?'. At the bottom of the form are two buttons: 'Volver' (Return) and 'Agregar lectura' (Add reading). In the top right corner, a green notification box with a checkmark icon displays the message '¡Libro Rayuela agregado correctamente!' (Book Rayuela added correctly!).

Editar un libro

Para la edición de una entidad, los pasos a realizar son:

1. Obtener la entidad (podemos utilizar nuestra funcionalidad ya creada de “Seleccionar libro”)
2. Cargar los datos de la entidad en los *input*, para que el usuario pueda editarla como desee.
3. Guardar cambios y actualizar.

Agregaremos un botón a BookDetails para que muestre u oculte el formulario de NewBook debajo, que es donde vamos a realizar la edición.

Nota: sería un buen momento de cambiar el nombre del formulario, de NewBook a BookForm, ya que no solo se utilizará para agregar libros sino también para editarlos.

Disponible

Las dos torres
J.R.R. Tolkien
★★★★★
352 páginas

Síntesis: La Compañía se ha disuelto y sus integrantes emprenden caminos separados. Frodo y Sam avanzan solos en su viaje a lo largo del río Anduin, perseguidos por la sombra misteriosa de un ser extraño que también ambiciona la posesión del Anillo. Mientras, hombres, elfos y enanos se preparan para la batalla final contra las fuerzas del Señor del Mal.

Ocultar formulario

Volver a la página principal

Título: Ingresar título

Autor: Ingresar autor

Puntuación: Ingresar cantidad de estrellas

Cantidad de páginas: Ingresar cantidad de páginas

URL de imagen: Ingresar url de imagen

¿Disponible? ☐

Volver Agregar lectura

```

51 | <Row>
52 |   <Button className="mb-2 me-2" variant="secondary" onClick={handleShowBookForm}>
53 |     {showBookForm ? "Ocultar formulario" : "Editar libro"}
54 |   </Button>
55 |   <Button className="me-2" onClick={clickHandler}>
56 |     Volver a la página principal
57 |   </Button>
58 | </Row>

```

Al hacer click en editar libro, vamos a cambiar un estado (llamemoslo *showEditForm*) que una vez que cambie a verdadero, muestre BookForm debajo de BookDetails.

Ahora, el formulario está preparado para solo agregar lecturas, ¿Cómo lo podemos convertir en un formulario de edición?

Primero, vamos a enviarle una prop (seteada por default en falso) a BookForm, que le informe si estamos usando el formulario para editar o para agregar

```

<Button variant="primary" type="submit">
  {isEditing ? "Editar lectura" : "Agregar lectura"}
</Button>

```

Paso siguiente, tenemos que precargarle los datos del libro en el formulario. Esto se lo enviamos por props (supongamos una prop llamada *book* o *entity*) y, utilizando el valor *value*, lo vamos llenando:

```

7  const BookForm = ({
8      book,
9      onBookAdded,
10     isEditing = false
11 }) => {
12     const [title, setTitle] = useState(book?.title);
13     const [author, setAuthor] = useState(book?.author);
14     const [rating, setRating] = useState(book?.rating);
15     const [pageCount, setPageCount] = useState(book?.pageCount);
16     const [imageUrl, setImageUrl] = useState(book?.imageUrl);
17     const [available, setAvailable] = useState(book?.available);
18

```



Título	Autor
<input type="text" value="Las dos torres"/>	<input type="text" value="J.R.R. Tolkien"/>
Puntuación	Cantidad de páginas
<input type="text" value="5"/>	<input type="text" value="352"/>
URL de imagen	
<input type="text" value="https://m.media-amazon.com/images/I/A1y0jd28riL_AC_UF1000,1000_QL80.jpg"/>	
<input checked="" type="checkbox"/> ¿Disponible?	
<input type="button" value="Volver"/> <input type="button" value="Editar lectura"/>	

Allí, utilizamos el operador ? para evitar un error en caso de que *book* sea nulo (para los casos donde queremos agregar un libro).

Finalmente, debemos crear un método que efectúe el guardado, actualización en la base de datos y actualización del lado cliente de la información provista. El método deberá ejecutarse de manera condicional, ya que si no está editando, deseamos que el método de agregado se lleve a cabo.

```

64     const handleSaveBook = (event) => {
65         event.preventDefault();
66
67         const bookData = {
68             title,
69             author,
70             rating: parseInt(rating, 10),
71             pageCount: parseInt(pageCount, 10),
72             imageUrl,
73             available
74         };
75
76         fetch(`http://localhost:3000/books/${book.id}`, {
77             headers: {
78                 "Content-type": "application/json"
79             },
80             method: "PUT",
81             body: JSON.stringify(bookData)
82         })
83             .then(res => res.json())
84             .then(() => {
85                 onBookSaved(bookData);
86             })
87             .catch(err => console.log(err))
88     }
89

```

onBookSaved es una callback que enviaremos desde el componente BookDetails, para poder obtener los nuevos campos actualizados y modificar **en tiempo real**, los detalles del libro.

Para ello, vamos a cambiar la lógica de BookDetails de manera que el libro que antes obteníamos directamente de *location.state*, ahora lo seteamos en un estado para su supervisión durante el ciclo de vida del componente.

```

const BookDetails = () => {
  const [showBookForm, setShowBookForm] = useState(false);
  const [book, setBook] = useState(null);

  const location = useLocation();
  const { id } = useParams();
  const navigate = useNavigate();

  useEffect(() => {
    const bookState = {
      ...location.state.book,
      id: parseInt(id, 10)
    }
    setBook(bookState)
  }, [location.state.book, id])
}

```

Lo que haremos allí entonces es un *useEffect* con dependencias. Ambas dependencias igual, solo deberían cambiar cuando el path cambie (en este caso, cuando el *state.book* cambie o el id de los *params*) Es decir, funcionará de manera similar a si no tuviera nada en el arreglo de dependencias.

Podemos chequear que el *useEffect* se ejecute realmente cuando deseemos con un *console.log* dentro de la *callback*.

```

32   const handleBookUpdated = (book) => {
33     |     setBook(book);
34   }

```

A su vez, debemos cambiar todas las referencias a los valores originales por *book?*.

```

48     <Card.Body>
49       <div className="mb-2">
50         {book?.available ?
51           <Badge bg="success">Disponible</Badge>
52           :
53           <Badge bg="danger">Reservado</Badge>
54         }
55       </div>
56       <Card.Title>{book?.title}</Card.Title>
57       <Card.Subtitle>{book?.author}</Card.Subtitle>
58       {ratingStars}
59       <p>{book?.pageCount} páginas</p>
60       <p className="my-3">
61         <b>Sinopsis</b>: {book?.summary}
62       </p>
63       <Row>
64         <Button className="mb-2 me-2" variant="secondary" onClick={handleShowBookForm}>
65           {showBookForm ? "Ocultar formulario" : "Editar libro"}
66         </Button>
67         <Button className="me-2" onClick={clickHandler}>
68           Volver a la página principal
69         </Button>
70       </Row>
71     </Card.Body>
72   </Card>
73   {showBookForm ? <BookForm isEditing={book === book} onBookSaved={handleBookUpdated} /> : null}

```

Si volvemos a Dashboard, observamos que el libro que editamos no cambia ¿Por qué sucede esto? Resulta que como nos manejamos dentro de Router, la aplicación no vuelve a montar el componente Dashboard una vez realizado (Router nunca realiza una recarga de la aplicación durante su ciclo de vida)

Una forma sencilla de arreglarlo es agregarle la dependencia de *location* al *useEffect* donde se hace el pedido de los libros. El inconveniente aquí es que **cada vez que cambie la *location* (el path) va a efectuarse el pedido al servidor**. Eso sería un gasto de recursos innecesario, porque no todas las rutas necesitan la información de libros disponibles.

Con un sencillo condicional entonces, nos aseguramos de que solo se realice el pedido en el path deseado:

```

19   useEffect(() => {
20     if (location.pathname === "/library") {
21       fetch("http://localhost:3000/books")
22         .then(res => res.json())
23         .then(data => setBookList([...data]))
24         .catch(err => console.log(err));
25     }
26   }, [location])
27
28

```


Con un *console.log* podemos comprobar que esto se cumpla.
Agregar mensajes de error y de éxito correspondientes para la actualización de libros.

Ejercicio de clase - eliminar un libro

Como desafío, armar la conexión entre el método armado de DELETE la entidad libro y la función *handleDeleteBook* en el lado cliente.

Nota: la respuesta de este método no es un JSON, sino un texto simple. Es decir, la callback que realiza el *res.json()* no es necesaria.

Fecha	Versionado actual	Autor	Observaciones
-------	-------------------	-------	---------------

25/05/2023	1.0.0	Gabriel Golzman	Primera versión
03/06/2023	1.0.1	Gabriel Golzman	Cambio encabezado
16/09/2023	1.1.0	Gabriel Golzman	Cambiado MockAPI por librería fake-api
29/09/2023	1.2.0	Gabriel Golzman	Agregada sección de <i>useReducer</i>
17/01/2024	2.0.0	Gabriel Golzman	Segunda versión. con .NET
28/01/2025	3.0.0	Gabriel Golzman	Tercera versión