

Universidad Tecnológica Nacional

Facultad Regional Rosario

Tecnicatura Universitaria de la Programación



Laboratorio de Computación III

Unidad 2.1

Manejo de state en React

Introducción	3
Manejo de eventos	3
Cómo React ejecuta las funciones de los componentes	4
state en React	5
Algunas aclaraciones más de useState()	6
Agregando nuevos libros mediante formularios	7
Escuchando el input de los usuarios	8
Manejar los datos ingresados por el usuario	11
Limpieza de los input y el concepto de two-way binding	11
Pasar data del componente hijo al componente padre	12
Ejercicio de clase	14

Introducción

Hasta este momento, nuestro mini-proyecto creado no es muy interesante para el usuario, ya que solo nos provee valores estáticos que no podemos interactuar o cambiar. Por ejemplo, no podemos agregar nuevos libros que leímos ni borrar los que ya vienen de nuestro código. En las siguientes secciones veremos cómo manejar eventos e interacciones con el usuario para darle dinamismo a nuestra aplicación.

Manejo de eventos

Vamos a agregar un botón que cambie el título del libro que estamos leyendo. Agregamos un componente de *bootstrap* Button al jsx del componente "BookItem", abajo del contador de páginas.

A todos los elementos de jsx que hacen de elementos de HTML se le pueden pasar como *props* los manejadores de eventos que su contraparte de HTML posee (pensemos en eventos como *onClick*, *onFocus*). Entonces, podemos agregar al componente Button un atributo *onClick*.

Pero, ¿Cómo lo escribimos? A *onClick* le vamos a pasar una **función** entre las llaves, ya que son atributos que esperan ejecutar código ante interacción del usuario. Escribiremos entonces:

```
<Button onClick={() => console.log("clicked!")}>
  Actualizar título
</Button>
```

Ese código nos muestra que cada vez que hacemos click en el botón, en consola imprimirá "Clicked!!". Lo comprobamos apretando F12 en el navegador.

Recordemos que es ideal que no tengamos demasiado código en nuestro jsx, por eso moveremos esta lógica a una variable más arriba y luego haremos referencia directamente a la misma en el botón:

```
const clickHandler = () => {
  console.log("clicked!");
};
```

```
<Button onClick={clickHandler}>Actualizar título</Button>
```

Un par de cosas a remarcar:

- Dentro del jsx, donde hacemos referencia a la función *clickHandler* no ponemos paréntesis. ¿Por qué no? Porque si los pusiéramos, el compilador interpreta **que en**

el momento que la lee debe ejecutarla, y no cuando realiza click. De esta manera, solo hacemos la referencia, y se ejecuta sólo cuando el usuario hace click.

- Es buena práctica siempre que se manejan eventos de poner *xxxxHandler* como nombre de función, donde xxxx representa lo que nosotros queramos, pero siempre lo terminamos en *Handler* para dar a entender que esta función está relacionada a eventos y no a otra cosa. Esto es, de todas formas, opcional.

Cómo React ejecuta las funciones de los componentes

Declaramos dentro del componente una variable llamada *titleUpdate* con *let*, y la igualamos a *title*. Luego, dentro de nuestro *clickHandler* escribiremos la siguiente línea:

```
titleUpdate = 'Actualizado!';
```

Y en el *jsx*, cambiaremos *title* por *titleUpdate*. Vemos que en pantalla, al apretar el botón y disparar el método *clickHandler*, no sucede nada. Si chequeamos la consola, el *console.log('Clicked!!')* si es invocado. ¿Por qué no funciona nuestro cambio de variable? Sencillamente, **porque React no funciona así.**

Pensemos que nuestros componentes son **funciones** con la única particularidad que retornan **jsx**. Al ser funciones deben ser invocados en algún momento. Por ejemplo, "BookItem" es invocado en el componente "Books", luego de haber invocado el componente "BookCard". El componente "BookItem" a su vez convoca también al componente "BookCard", para luego invocar "ReadDate", dónde "ReadDate" ya devuelve *jsx* puro de React.

Todo esta búsqueda se conoce como búsqueda en profundidad dentro de un árbol de decisiones, donde la raíz del árbol es el componente "App", que se invoca con las líneas:

```
const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
);
```

Una vez que se ha renderizado en pantalla todo el árbol, React ha terminado su trabajo. Es decir, que no se encargará de realizar cambios en lo provisto en la pantalla.

A menos que nos aprovechemos del concepto de *state*.

state en React

Reflexionemos un poco sobre lo que queremos hacer:

1. Deseamos cambiar el título de los libros, de lo que sea que nos vino por *props* a la cadena de texto 'Actualizado!!'.
2. Para ello, debemos tomar lo que está en *title* y reemplazarlo de alguna manera.
3. Luego React debe re-renderizar el componente y mostrar la nueva cadena para que el usuario vea el cambio en su pantalla.

¡Parece un montón! Pero React nos proporciona, dentro de su librería, una función, del tipo *hook* o "gancho", llamada *useState*. Todos los *hooks* de React (y hay varios, además de que podemos crear los nuestros) comienzan con la palabra *use*. *useState* sólo puede ser llamada dentro del componente, **no** puede ser llamada fuera del mismo o dentro de una función anidada (como *clickHandler*).

```
1 | import { useState } from "react";
```

Entonces, dentro del componente llamaremos a *useState()*. *useState* recibe como parámetro el **valor inicial que queremos darle a una variable que creará React**. En este caso, le pasamos *title*.

¿Que retorna *useState*? Retorna un arreglo con dos valores, el primer valor es nuestra nueva variable ya inicializada y el segundo es una función **que nos servirá para setear el valor de dicha variable**. Podemos utilizar *array destructuring* para obtener ambos, ya que en *array destructuring* solo importa la posición de los elementos.

```
const [newTitle, setNewTitle] = useState(title);
```

Es común que el primer elemento tenga un nombre representativo de lo que es, y el segundo sea la palabra *set* seguida del nombre del primer elemento.

Entonces, dentro de nuestro *clickHandler* llamaremos a *setNewTitle* y le pasaremos como parámetro el nuevo título que deseamos actualizar, además de imprimirlo en consola en la misma función:

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

import { Button, Card } from "react-bootstrap";

const BookItem = ({ title, author, pageCount, rating, imageUrl }) => {

  const [newTitle, setNewTitle] = useState(title);

  const clickHandler = () => {
```

```

    console.log(newTitle);

    setNewTitle(";Actualizado!");
  };

  return (

    <Card className="mx-3" style={{ width: "22rem" }}>

      <Card.Img

        height={400}

        variant="top"

        src={imageUrl ?? "https://bit.ly/47NylZk"}

      />

      <Card.Body>

        <Card.Title>{newTitle}</Card.Title>

        <Card.Subtitle>{author}</Card.Subtitle>

        <div>{rating?.length} estrellas</div>

        <p>{pageCount} páginas</p>

        <Button onClick={clickHandler}>Actualizar título</Button>

      </Card.Body>

    </Card>

  );
};

export default BookItem;

```

Ahora si vemos que en pantalla nos cambian los textos de los títulos. Si nos fijamos en la consola, ahora nos imprime el valor viejo de la variable, siendo que la línea de `console.log` está después de `setNewTitle()`. Esto es debido a que la función no es llamada automáticamente sino asincrónicamente, es decir, se coloca en una cola para ser llamada posteriormente, mientras que `console.log()` se ejecuta luego de ser leída.

Entonces, para poder cambiar la data que posee un componente **debemos recurrir si o si al `state`, a variables guardadas dentro de la memoria de React**. Nunca podremos hacer cambios directos en los componentes sin `state`.

Algunas aclaraciones más de `useState()`

- Cada `state` es particular para cada componente, es decir, el cambio de título que yo realizo en uno de los `BookItem` no se refleja en los otros `BookItem`. Además de eso, React solo hará la comparación y el re-renderizado de ese componente y no de los otros. Esto lo podemos probar agregando la siguiente línea al componente:

```
console.log("BookItem evaluado por React");
```

Luego, si refrescamos la página, se imprimirá 4 veces la línea (cada vez que se renderiza un componente) pero al apretar uno de los botones, solo se imprimirá una vez.

- Al `state` lo declaramos como `const` siendo que su primer valor del `array` va a estar cambiando constantemente ¿Por qué? Porque no podemos, por ejemplo, decir `newTitle = 'Nuevo título'` en nuestro código, pero sí podemos utilizar la función `setTitle('Nuevo título')` que se encargará por detrás de realizar los pasos necesarios para mutar el arreglo.
- La llamada a `setTitle()` dispara un nuevo renderizado del componente, pero `useState` no es llamado para reemplazar el título otra vez, ya que React sabe que hubo un cambio de estado desde la creación del componente, de manera que no lo reemplaza con el valor inicial que le aportamos, sino con el valor enviado por `setTitle()`.

Agregando nuevos libros mediante formularios

Vamos a concentrarnos ahora en la creación de un formulario para que el usuario pueda registrar los nuevos libros que va leyendo. Esto nos permitirá aprender más sobre el `state` en React y cómo trabajar con la data provista a la aplicación.

Crearemos una nueva carpeta dentro de `components` llamada `NewBook` donde crearemos el componente `NewBook` junto a su estilizado:

```
import { Button, Card, Col, Form, Row } from "react-bootstrap";
```

```

const NewBook = () => {

  return (

    <Card className="m-4 w-50" bg="success">

      <Card.Body>

        <Form className="text-white">

          <Row>

            <Col md={6}>

              <Form.Group className="mb-3" controlId="bookTitle">

                <Form.Label>Título</Form.Label>

                <Form.Control type="text" placeholder="Ingresar título"
/>

              </Form.Group>

            </Col>

            <Col md={6}>

              <Form.Group className="mb-3" controlId="bookAuthor">

                <Form.Label>Autor</Form.Label>

                <Form.Control type="text" placeholder="Ingresar autor"
/>

              </Form.Group>

            </Col>

          </Row>

          <Row>

            <Col md={6}>

              <Form.Group className="mb-3" controlId="bookRating">

                <Form.Label>Puntuación</Form.Label>

                <Form.Control

```



```

        type="number"

        placeholder="Ingresar cantidad de estrellas"

        max={5}

        min={0}

    />

</Form.Group>

</Col>

<Col md={6}>

    <Form.Group className="mb-3" controlId="bookPageCount">

        <Form.Label>Cantidad de páginas</Form.Label>

        <Form.Control

            type="number"

            placeholder="Ingresar cantidad de páginas"

            min={1}

        />

    </Form.Group>

</Col>

</Row>

<Row className="justify-content-between">

    <Form.Group className="mb-3" controlId="bookImageUrl">

        <Form.Label>URL de imagen</Form.Label>

        <Form.Control type="text" placeholder="Ingresar url de
imagen" />

    </Form.Group>

</Row>

```

```

        <Row className="justify-content-end">
          <Col md={3} className="d-flex justify-content-end">
            <Button variant="primary" type="submit">
              Agregar lectura
            </Button>
          </Col>
        </Row>
      </Form>
    </Card.Body>
  </Card>
);
};

export default NewBook;

```

Veamos qué maqueta de JSX armamos allí:

- Primero encerramos el form dentro de una tarjeta, con la variante *success* (color verde).
- Luego declaramos el componente Form, seguido por controladores para los diferentes valores de la entidad libro. Cada controlador estará encerrado dentro de su FormGroup, poseerá un Label y al finalizar el formulario poseemos el botón de *submit*.
- A su vez, agregamos el concepto de [bootstrap grid](#) para obtener por completo los beneficios *responsive* de dicha librería. En este caso, organizamos el formulario en 4 filas y las primeras dos filas, organizadas en dos columnas iguales.

Agregamos el componente NewBook a App, modificando el *className* del div principal para obtener el siguiente resultado:

```

<div className="d-flex flex-column align-items-center">
  <h2>Books Champion App</h2>

```

```
<p>¡Quiero leer libros!</p>

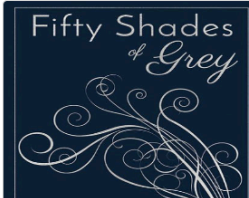



<NewBook />

<Books books={books} />

</div>
```

Books Champion App
¡Quiero leer libros!

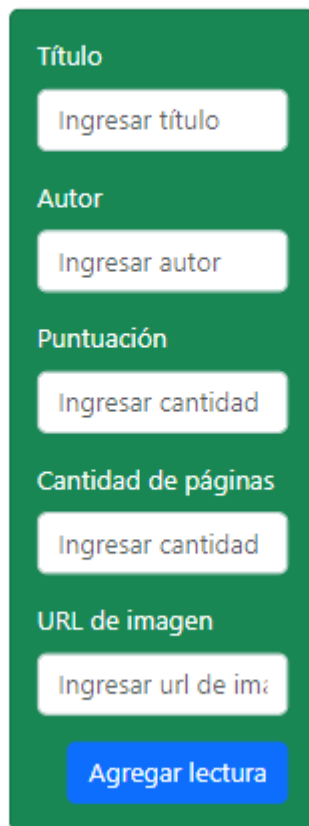
Título	Autor
<input type="text" value="Ingresar título"/>	<input type="text" value="Ingresar autor"/>
Puntuación	Cantidad de páginas
<input type="text" value="Ingresar cantidad de estrellas"/>	<input type="text" value="Ingresar cantidad de páginas"/>
URL de imagen	
<input type="text" value="Ingresar url de imagen"/>	
<input type="button" value="Agregar lectura"/>	



En pantallas más pequeñas, nuestro diseño se ve así:

Books Champion App

¡Quiero leer libros!



Título

Ingresar título

Autor

Ingresar autor

Puntuación

Ingresar cantidad

Cantidad de páginas

Ingresar cantidad

URL de imagen

Ingresar url de imagen

Agregar lectura

Escuchando el *input* de los usuarios

Ahora bien, es de nuestro interés siempre estar al tanto de los cambios producidos en estos bloques *input*. Para ello nosotros podemos agregar ciertas *props* nativas de jsx, de manera que se encarguen de disparar una función que deseemos ante una interacción del usuario, similar a cuando el mismo hacía click sobre el botón "cambiar título".

Entonces:

```
<Form.Control
  type="text"
  placeholder="Ingresar título"
  onChange={changeTitleHandler}
/>
```

La función *changeTitleHandler* es una función que se ejecutará cada vez que ingresemos una letra, número o seleccionemos algo en un *dropdown*. Podríamos también utilizar la *prop onInput* pero ella sólo toma en cuenta los valores ingresados por teclado, y no los seleccionables.

Todas las *props* nativas de *jsx* que invocan un evento, adhieren automáticamente el objeto *event* a la función asociada. Es decir, cada vez que se dispara un evento en el navegador, a nosotros se nos pasa como parámetro información sobre él mismo. Podemos verlo de la siguiente manera:

```
4  ✓  const changeTitleHandler = (event) => {
5      console.log(event);
6  }
```

El objeto *event* nos aporta muchísima información sobre el evento sucedido (en este caso, el *input* de un valor de texto por el usuario), pero lo que nos interesa a nosotros es lo que se encuentra dentro de *target*, en la propiedad *value*. Haremos *console.log* del mismo para verlo:

```
4  ✓  const changeTitleHandler = (event) => {
5      console.log(event.target.value);
6  }
```

Como observamos, *event.target.value* va a guardar siempre **el último valor que se encuentra en el *input* en este momento**. De esta manera, siempre podremos acceder a lo último que ingresó el usuario.

Ahora, para poder llevar un registro interno de este valor en el componente, utilizaremos de vuelta el *hook useState* de React, de la siguiente manera:

```
5  const [enteredTitle, setEnteredTitle] = useState('');
6  // El valor inicial son las comillas vacías
```

Y la línea de *console.log* dentro de la función *changeTitleHandler*, la cambiaremos por el *setEnteredTitle*:

```
import { useState } from "react";
import { Button, Card, Col, Form, Row } from "react-bootstrap";

const NewBook = () => {
  const [enteredTitle, setEnteredTitle] = useState("");
  const [enteredAuthor, setEnteredAuthor] = useState("");
  const [enteredRating, setEnteredRating] = useState("");
  const [enteredPageCount, setEnteredPageCount] = useState("");
  const [enteredImageUrl, setEnteredImageUrl] = useState("");

  const changeTitleHandler = (event) => {
    setEnteredTitle(event.target.value);
  };
};
```

```

const changeAuthorHandler = (event) => {
  setEnteredAuthor(event.target.value);
};

const changeRatingHandler = (event) => {
  setEnteredRating(event.target.value);
};

const changePageCountHandler = (event) => {
  setEnteredPageCount(event.target.value);
};

const changeImageUrlHandler = (event) => {
  setEnteredImageUrl(event.target.value);
};

return (
  <Card className="m-4 w-50" bg="success">
    <Card.Body>
      <Form className="text-white">
        <Row>
          <Col md={6}>
            <Form.Group className="mb-3" controlId="bookTitle">
              <Form.Label>Título</Form.Label>
              <Form.Control
                type="text"
                placeholder="Ingresar título"
                onChange={changeTitleHandler}
                value={enteredTitle}
              />
            </Form.Group>
          </Col>
          <Col md={6}>
            <Form.Group className="mb-3" controlId="bookAuthor">
              <Form.Label>Autor</Form.Label>
              <Form.Control
                onChange={changeAuthorHandler}
                type="text"
                placeholder="Ingresar autor"
              />
            </Form.Group>
          </Col>
        </Row>
      </Form>
    </Card.Body>
  </Card>
);

```

```

        <Col md={6}>
          <Form.Group className="mb-3" controlId="bookRating">
            <Form.Label>Puntuación</Form.Label>
            <Form.Control
              type="number"
              onChange={changeRatingHandler}
              placeholder="Ingresar cantidad de estrellas"
              max={5}
              min={0}
            />
          </Form.Group>
        </Col>
        <Col md={6}>
          <Form.Group className="mb-3" controlId="bookPageCount">
            <Form.Label>Cantidad de páginas</Form.Label>
            <Form.Control
              onChange={changePageCountHandler}
              type="number"
              placeholder="Ingresar cantidad de páginas"
              min={1}
            />
          </Form.Group>
        </Col>
      </Row>
      <Row className="justify-content-between">
        <Form.Group className="mb-3" controlId="bookImageUrl">
          <Form.Label>URL de imagen</Form.Label>
          <Form.Control
            onChange={changeImageUrlHandler}
            type="text"
            placeholder="Ingresar url de imagen"
          />
        </Form.Group>
      </Row>
      <Row className="justify-content-end">
        <Col md={3} className="d-flex justify-content-end">
          <Button variant="primary" type="submit">
            Agregar lectura
          </Button>
        </Col>
      </Row>
    </Form>
  </Card.Body>

```

```

    </Card>
  );
};

export default NewBook;

```

Vemos que inicializamos a cada uno de los estados con una cadena de texto vacía. Esto es debido a que los valores obtenidos en elementos *input* siempre van a ser guardados en el objeto *event* como *strings*, incluso si el valor guardado es un número.

Manejar los datos ingresados por el usuario

Buscamos entonces agregar un libro cuando el usuario ingresa todos los campos y clickea el botón de "Agregar lectura". Podríamos estar tentados a agregar una *prop* de *onClick* al botón y manejarlo de esa forma, pero en realidad lo correcto sería utilizar las bondades del elemento *form* en donde se encuentra el botón y agregarle la *prop onSubmit* al mismo, de la siguiente forma:

```

<Form className="text-white" onSubmit={submitBookHandler}>

```

submitBookHandler será nuestra función a cargo de realizar el *submit* justamente.

```

const submitBookHandler = (event) => {
  event.preventDefault();
  const bookData = {
    bookTitle: enteredTitle,
    bookAuthor: enteredAuthor,
    bookRating:
      enteredRating !== ""
        ? Array(parseInt(enteredRating, 10)).fill("*")
        : Array(0),
    pageCount: parseInt(enteredPageCount, 10),
    imageUrl: enteredImageUrl,
  };

  console.log(bookData);
};

```

Analicemos un poco la función anterior:

- `event.preventDefault()` previene que se dispare un evento automático, en el cuál la página luego de realizar `submit` **vuelve a recargarse**. Como no queremos realizar un `reload`, `preventDefault()` nos permite evitar ese comportamiento.
- Luego, crearemos un objeto con los valores **actuales** del formulario. Para `pageCount` y `rating`, realizamos modificaciones necesarias para adecuarse a los requisitos de `BookItem`.
- Luego, comprobamos que el objeto se creó correctamente en la consola.

Limpieza de los *input* y el concepto de *two-way binding*

¿Cómo podemos hacer para limpiar el contenido de los *input* luego de que el usuario agrega la lectura, por si quiere agregar nuevas? Podríamos directamente llamar a las funciones `set` con una cadena vacía, pero eso **no haría que el valor del input que se muestra cambie**.

Utilizaremos en realidad, el concepto de *two-way binding*. El mismo refiere a que lo que yo ingreso en el *input* se refleja en el código, y lo que está en el código yo lo reflejo en el *input* (es decir, lo muestro siempre en pantalla).

Para lograr esto, sencillamente agregaremos la *prop value* a nuestro *input* de *title* de la siguiente manera:

```
value = {enteredTitle}
```

y luego en nuestra función de `submitBookHandler` agregamos la línea:

```
setEnteredTitle("");
```

Y probamos a ver si el *input* se limpia o no. Realizamos lo mismo con el autor, la cantidad de páginas y la fecha.

Pasar data del componente hijo al componente padre

Hasta el momento, vimos que es posible pasar data del componente padre al hijo mediante la utilización de *props*, pero, ¿Cómo realizamos el camino inverso? En el caso de que quisiéramos pasar, por ejemplo, la data del nuevo libro al componente "App", para que lo muestre junto a las otras lecturas ¿Cómo hacemos?

Primero, nos situaremos en App. Allí pasaremos como *props* lo siguiente:

```
<NewBook onBookDataSaved={saveBookDataHandler} />
```

La *prop onBookDataSaved* contendrá a la función `saveBookDataHandler`, la cual se encargará de recibir el objeto con el nuevo libro, copiar los contenidos de ese objeto en otro, que además le agregue un *id random* (con la función `Math.random()` (que no es ideal en

este caso, pero sirve como ejemplo)) y luego, un `console.log()` donde podremos ver que todo ande correctamente.

```

5  ✓  const saveBookDataHandler = (enteredBookData) => {
6  ✓      const bookData = {
7          ...enteredBookData,
8          id: Math.random().toString(),
9      };
10     console.log(bookData);
11 };
```

Todo genial, pero ¿De dónde viene el parámetro `enteredBookData`? Eso es justamente lo que nos va a aportar el componente hijo. Nos dirigimos a `NewBook`. Allí podremos realizar el *destructuring* de las *props* para obtener la función `onBookDataSaved` e invocarla dentro de la función que refiere al `submit`.

```

const submitBookHandler = (event) => {
    event.preventDefault();
    const bookData = {
        bookTitle: enteredTitle,
        bookAuthor: enteredAuthor,
        bookRating:
            enteredRating !== ""
                ? Array(parseInt(enteredRating, 10)).fill("*")
                : Array(0),
        pageCount: parseInt(enteredPageCount, 10),
        imageUrl: enteredImageUrl,
    };

    onBookDataSaved(bookData);
    setEnteredTitle("");
    setEnteredAuthor("");
    setEnteredRating("");
    setEnteredPageCount("");
    setEnteredImageUrl("");
};
```

Ahora, comprobamos su funcionamiento en pantalla, chequeando que el objeto imprima la *id* dentro al objeto nuevo creado en consola.

Repasemos el camino realizado, ya que es el mismo siempre que queramos pasar *data* de hijo a padre:

1. Pasamos por *props* una función que declaramos en el componente padre.

2. Esa función va a recibir un parámetro, el cual va a ser enviado por el componente hijo en el momento que se invoque.
3. Agregar la invocación de la función en el componente hijo.

Ejercicio de clase

Deseamos colocar por encima de la lista de libros, un elemento *p* que nos muestre que libro fue seleccionado, para ello:

Pasos:

1. Cambiar el texto del botón para actualizar el título a “Seleccionar libro”.
2. Luego, escribir una función que suba desde BookItem a Books, el título del libro seleccionado.
3. Mostrar el valor en el componente Books.

Fecha	Versionado actual	Autor	Observaciones
12/04/2022	1.0.0	Gabriel Golzman	Primera versión
24/08/2022	1.0.1	Gabriel Golzman	Fix de una imagen
23/11/2023	2.0.0	Gabriel Golzman	Segunda versión
30/11/2023	2.0.1	Gabriel Golzman	Correcciones generales