

Clase 20 - React - Componentes

Renderizado de múltiples elementos en un Componente - Etiqueta Fragment

En React, un fragment es una forma de renderizar múltiples elementos hijos sin tener que crear un elemento padre adicional en el DOM ya que React requiere que solo exista un solo elemento padre en cada componente. Un fragment es como un contenedor virtual que permite agrupar elementos y componentes sin agregar nodos HTML adicionales a la página.

En React, cuando necesitamos renderizar varios elementos hijos dentro de un componente, normalmente los envolvemos dentro de un elemento padre, como un `<div>`. Por ejemplo:

```
function MiComponente() {  
  return (  
    <div>  
      <h1>Título del componente</h1>  
      <p>Este es el contenido del componente.</p>  
    </div>  
  );  
}
```

En este ejemplo, estamos utilizando un elemento `<div>` para agrupar los elementos `<h1>` y `<p>` y crear una estructura HTML válida. Sin embargo, en algunos casos, esto puede no ser deseable o puede interferir con la estructura o el estilo de la página.

En estos casos, podemos utilizar un fragment para evitar tener que agregar un elemento padre adicional. Para utilizar un fragment, se puede utilizar la sintaxis `<React.Fragment>` o la sintaxis abreviada `<>`. Por ejemplo:

```
function MiComponente() {  
  return (  
    <>  
      <h1>Título del componente</h1>  
      <p>Este es el contenido del componente.</p>  
    </>  
  );  
}
```

En este ejemplo, estamos utilizando un fragment para agrupar los elementos `<h1>` y `<p>` sin crear un elemento padre adicional. Cuando se renderice el componente en la página, los elementos `<h1>` y `<p>` se

mostrarán sin estar envueltos en un elemento `<div>`.

Los fragmentos son una herramienta útil para crear componentes más limpios y organizados en React, especialmente cuando se trabaja con estructuras de componentes complejas o componentes que necesitan renderizar varios elementos hijos.

Los fragmentos son necesarios en React por varias razones:

- Evitar elementos innecesarios en el DOM: Cuando renderizamos varios elementos dentro de un componente en React, normalmente necesitamos envolverlos en un elemento padre para crear una estructura HTML válida. Sin embargo, a veces esto puede interferir con la estructura o el estilo de la página. Al utilizar fragmentos, podemos agrupar los elementos sin tener que agregar un elemento padre adicional en el DOM, lo que ayuda a mantener el HTML más limpio y organizado.
- Mejorar el rendimiento: Agregar elementos innecesarios al DOM puede afectar el rendimiento de la página, especialmente en aplicaciones con una gran cantidad de componentes y elementos. Al utilizar fragmentos, podemos reducir la cantidad de elementos en el DOM y mejorar el rendimiento de la página.
- Trabajar con listas y componentes dinámicos: Al renderizar listas o componentes dinámicos en React, a menudo necesitamos crear varios elementos dentro de un componente. Al utilizar fragmentos, podemos agrupar los elementos sin tener que agregar un elemento padre adicional para cada elemento en la lista, lo que ayuda a mantener el código más legible y organizado.

En resumen, los fragmentos son una herramienta útil en React para agrupar elementos y componentes sin tener que agregar elementos innecesarios al DOM y mejorar el rendimiento de la página.

¿Qué son los Hooks (Ganchos)?

Los Hooks son una nueva incorporación en React 16.8.0 por el año 2018. Los Hooks son funciones que permiten "engancharse" al estado de React y el ciclo de vida desde componentes de función. Los hooks se usan para agregar características como estado, efectos secundarios y contexto a los componentes funcionales de React. Antes de la introducción de los hooks, estos tipos de características sólo estaban disponibles en componentes de clase.

Algunos de los hooks que nos provee react para componentes funcionales son:

- **useState**: este hook permite a los componentes funcionales de React tener estado. Con `useState`, puedes declarar una variable de estado y una función para actualizarla dentro de un componente funcional.
- **useEffect**: este hook permite a los componentes de React realizar efectos secundarios (como llamadas a API) después de la renderización del componente.
- **useContext**: este hook permite a los componentes de React acceder al contexto definido en un componente superior sin necesidad de pasar props a través de componentes intermedios.

Los hooks permiten a los desarrolladores de React escribir componentes más simples y fáciles de entender, sin sacrificar la funcionalidad. Además, los hooks se pueden reutilizar en diferentes componentes de React, lo que hace que el código sea más modular y fácil de mantener.

Este ejemplo renderiza un contador. Cuando hacemos click en el botón, incrementa el valor:

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

function Example() {
  // Declara una nueva variable de estado, que llamaremos "count", que permita
  // mantener el estado de la variable count entre las distintas renderizaciones del
  // componente
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

En el código anterior, **useState** es un *Hook*. Lo llamamos dentro de un componente de función para agregarle un estado local. React mantendrá este estado entre re-renderizados. `useState` devuelve un par: el valor de estado actual y una función que le permite actualizarlo.

El componente `Example` tiene una variable de estado llamada `count`, que se inicializa con un valor de 0 mediante el `useState` hook.

El componente renderiza un `div` que contiene un párrafo que muestra el valor actual de `count` y un botón que actualiza el valor de `count` cuando se hace clic en él. El botón llama a la función `setCount` cuando se hace clic, y le pasa como argumento el valor actual de `count` más 1. La función `setCount` actualiza el valor de `count` en el estado interno del componente y provoca una nueva renderización del componente.

En resumen, este ejemplo demuestra cómo utilizar el hook `useState` para agregar estado a un componente funcional de React y cómo actualizar ese estado en respuesta a eventos del usuario.

El único argumento para `useState` es el estado inicial. En el ejemplo anterior, es 0 porque nuestro contador comienza desde cero, pero podría ser un string, booleano, etc. El argumento de estado inicial solo se usa durante el primer renderizado.

Reglas para trabajar con Hooks

Hay tres reglas para trabajar con hooks:

- Los hooks solo pueden ser llamados desde componentes funcionales de React o desde otros hooks. No se pueden utilizar dentro de componentes de clase.
- Llamaremos Hooks solo en el nivel superior: No los llamaremos dentro de ciclos, condicionales o funciones anidadas. En cambio, siempre usaremos Hooks en el nivel superior de nuestra función en React, antes de cualquier retorno prematuro. Siguiendo esta regla, nos aseguramos de que los hooks se llamen en el mismo orden cada vez que un componente se renderiza. Esto es lo que permite a React preservar correctamente el estado de los hooks entre múltiples llamados a `useState` y `useEffect`.

- Los hooks no deben ser utilizados dentro de bucles o condiciones, ya que esto puede llevar a comportamientos inesperados y errores.

Ver más en <https://es.reactjs.org/docs/hooks-rules.html#explicación>

Hooks del ciclo de vida para componentes funcionales

En los componentes funcionales, los métodos de ciclos de vida toman dos parámetros, el primero es una función que contiene el código a ejecutar, y el segundo parámetro es una variable de dependencia. Se puede pasar sólo corchetes vacíos si queremos ejecutar la función solamente una vez cuando se carga la página.

- **useEffect(on Start):**- Este hook necesita ser importado desde react, y se ejecuta cuanto el componente se renderiza. Se puede utilizar este hook para la implementación de cualquier método que requiera ser llamado cuando se renderiza el componente. Hay que tener en cuenta que sólo se pasa corchetes vacíos como segundo parámetro para que funcione como tal.

```
//Definición resumida, si se pasa un array vacío, se ejecuta solo en el
primer renderizado
React.useEffect(() => {}, []);

//Definición con ejemplo, si se pasa un array vacío, se ejecuta solo en el
primer renderizado
React.useEffect(() => { //Similar a componentDidMount() en componente de
clase
  getCodigoTelefonicoPais(); //funcion que se va a invocar cuando se
renderiza por primera vez
}, []);
```

- **useEffect(on update)** Cuando necesitamos actualizar el estado de un componente, cuando se cambian los datos, podemos pasar entre corchetes las variables de dependencia que supuestamente cambian con el estado.

```
//Definición resumida para cualquier cambio
React.useEffect(() => {}); //Similar a componentDidUpdate en componentes de
clase

//Definición resumida para cambio de prop o state "empresa"
React.useEffect(() => {}, [empresa]); //Similar a componentDidUpdate con
variables en componentes de clase

//Definición con ejemplo
React.useEffect(() => { //Similar a componentDidUpdate con variable
  getResumenEmpresa();
}, [empresa]); //Esta función se va a ejecutar automáticamente cuando la
variable empresa cambia y el componente es actualizado
```

- **useEffect(on destroy):** este hook se utiliza cuando se necesita hacer una limpieza de variables del componente antes de que el mismo sea destruido. Se puede pasar un array vacío a la función como se

indica a continuación:

```
//Definición resumida
React.useEffect(() => { //Similar a ComponentWillUnmount()
  return () => {};
},[]);

//Definición con código de ejemplo
React.useEffect(() => { //Similar a ComponentWillUnmount()
  return () => {
    console.log("componente limpiado");
  };
}, []);
```

Paso de datos entre componentes

En React, todo es un componente, y una interfaz de usuario puede estar constituida por múltiples componentes interactuando entre sí, por lo que resulta necesario pasar información entre los componentes para comunicarse entre padre e hijos. React utiliza el objeto **props** para pasar datos desde un componente padre a un hijo cuando trabajamos con componentes funcionales. El objeto **props** contiene las propiedades que se pasan al componente que se va a renderizar.

```
function Usuario(props) {
  return <h1>Hola, {props.nombre}, bienvenido/a a Desarrollo de Software</h1>;
}
function App() {
  return (
    <div>
      <Usuario nombre="Sara" />
    </div>
  );
}
ReactDOM.render(<App />, document.getElementById('root'));
```

Estado de un componente

En React los estados permiten mantener trazabilidad de los cambios de variable o de cualquier cambio que gestiona el componente para re-renderizar. El estado es similar a cómo con **props** se mantiene un objeto en un componente. El estado sólo debe modificarse utilizando las funciones de React provistas a tal fin, como **useState()**, o en el caso del ejemplo a continuación con **setCantidadClick** para cambiar el valor del estado **cantidadClick**.

Se puede usar el estado previo utilizando arrow function dentro del setCantidadClick

```
import React from 'react';
import {useState} from 'react';
```

```
//Definición del componente funcional Button
const Button = () => {
  const [cantidadClick, setCantidadClick] = useState(0); //useState es un Hook
  de React que permmites añadir estado a un componente de función. En este caso
  asigna el valor inicial de 0
  return (
    <button
      type="button"
      className="my-button"
      onClick={() => setCantidadClick((prevCantidadClick) =>
        prevCantidadClick + 1 )}> {/* elemento button de html 5 */}
      Cantidad de veces clickeado el botón {cantidadClick}
    </button>
  );
}

function App() {
  return (
    <div className="App">
      <Button/> {/* Componente de función, no es un elemento de HTML!!! */}
      <Button/> {/* Este componente Button si bien es el mismo tipo de
componente que el de la línea de código previa, posee un estado propio e
independiente*/}
    </div>
  );
}
```

Nota: Se debe tener en cuenta que cuando declaramos una variable de estado utilizamos la función `useState()`

```
const [cantidadClick, setCantidadClick] = useState(0);
```

Los nombres *cantidadClick* y *setCantidadClick* no forman parte de la API de React. Lo que significa que *useState* devuelve un par de valores para las dos variables definidas, donde *cantidadClick* se establece en el primer valor devuelto por *useState* (es decir 0) y *setCantidadClick* es un puntero a función que *setState* devuelve para permitir actualizar el valor (como está en `onClick={() => setCantidadClick((prevCantidadClick) =>)`). Este concepto es de javascript y se llama desestructuración de arrays https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment#array_destructuring

El código anterior se podría haber escrito de la siguiente manera:

```
var cantidadClickEstado = useState(0); //Devuelve un array con 2 valores
var cantidadClick = cantidadClickEstado[0]; //Primer elemento del array, contiene
el valor inicial de 0 en este caso
var setCantidadClick = cantidadClickEstado[1]; //Segundo elemento contiene el
puntero a una función
```

```
elemento del array //para actualizar el primer
```

Cómo pasar información desde un componente hijo a un componente padre

En React se pueden utilizar funciones callback. En el componente padre se puede implementar una función que va estar escuchando el cambio.

```
//Componente Hijo
const Nombre = (props) => {
  return (
    <input name="firstName" onChange={props.onCambios} />
  );
}

//Componente padre
function App() {

  const onCambioEnHijoHandler = (event) => { //Parámetro en el que recibimos los
    valores del evento
    console.log("Valor modificado por:" + event.target.value) //del objeto event,
    obtenemos el componente, y del mismo, la propiedad value
  };

  return (
    <div className="App">
      <Nombre onCambios={onCambioEnHijoHandler} /> {/*Aqui pasamos el nombre
de la función definida en el padre que va a recibir los cambios del componente
hijo al producirse cambios */}
    </div>
  );
}
```

Renderizado condicional

React, podemos crear distintos componentes que encapsulan el comportamiento que necesitamos. Entonces, podemos renderizar solamente algunos de ellos, dependiendo del estado de nuestra aplicación.

El renderizado condicional en React funciona de la misma forma que lo hacen las condiciones en JavaScript. Usa operadores de JavaScript como if o el operador condicional para crear elementos representando el estado actual, y deja que React actualice la interfaz de usuario para emparejarlos.

Consideremos estos dos componentes:

```
function UserGreeting(props) {
  return <h1>Welcome back!</h1>;
}
```

```
function GuestGreeting(props) {  
  return <h1>Please sign up.</h1>;  
}
```

Vamos a crear un componente Greeting que muestra cualquiera de estos componentes dependiendo si el usuario ha iniciado sesión:

```
function Greeting(props) {  
  const isLoggedIn = props.isLoggedIn;  
  if (isLoggedIn) {  
    return <UserGreeting />;  
  }  
  return <GuestGreeting />;  
}  
  
const root = ReactDOM.createRoot(document.getElementById('root'));  
// Intentar cambiando isLoggedIn={true}:  
root.render(<Greeting isLoggedIn={false} />);
```

Variables de elementos

Podemos usar variables para almacenar elementos. Esto puede ayudarnos para renderizar condicionalmente una parte del componente mientras el resto del resultado no cambia.

Consideremos estos dos componentes nuevos que representan botones de cierre e inicio de sesión:

```
function LoginButton(props) {  
  return (  
    <button onClick={props.onClick}>  
      Login  
    </button>  
  );  
}  
  
function LogoutButton(props) {  
  return (  
    <button onClick={props.onClick}>  
      Logout  
    </button>  
  );  
}
```

En el siguiente ejemplo, crearemos un componente con estado llamado LoginControl.

El componente va a renderizar <LoginButton/> o <LogoutButton /> dependiendo de su estado actual. También va a renderizar un <Greeting /> del ejemplo anterior:


```
import React, { useState } from 'react';
import ReactDOM from 'react-dom';

function LoginControl() {
  const [isLoggedIn, setIsLoggedIn] = useState(false);

  const handleLoginClick = () => {
    setIsLoggedIn(true);
  };

  const handleLogoutClick = () => {
    setIsLoggedIn(false);
  };

  let button;
  if (isLoggedIn) {
    button = <LogoutButton onClick={handleLogoutClick} />;
  } else {
    button = <LoginButton onClick={handleLoginClick} />;
  }

  return (
    <div>
      <Greeting isLoggedIn={isLoggedIn} />
      {button}
    </div>
  );
}
```

If en una línea con operador lógico &&

Podemos incluir expresiones JS en JSX envolviéndolas en llaves. Esto incluye el operador lógico && de JavaScript. Puede ser útil para incluir condicionalmente un elemento:

```
function Mailbox(props) {
  const unreadMessages = props.unreadMessages;
  return (
    <div>
      <h1>Hello!</h1>
      {unreadMessages.length > 0 &&
        <h2>
          You have {unreadMessages.length} unread messages.
        </h2>
      }
    </div>
  );
}
```

```
const messages = ['React', 'Re: React', 'Re:Re: React'];

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Mailbox unreadMessages={messages} />);
```

En JavaScript, **true && expresión** siempre evalúa a **expresión**, y **false && expresión** siempre evalúa a **false**.

Por eso, si la condición es true, el elemento justo después de && aparecerá en el resultado. Si es false, React lo ignorará.

Se debe tener en cuenta que retornar expresiones falsas hará que el elemento después de ' && ' sea omitido pero retornará el valor falso.

If-Else en una línea con operador condicional

Otro método para el renderizado condicional de elementos en una línea es usar el operador condicional condición ? true : false de JavaScript (Operador ternario).

En el siguiente ejemplo, lo usaremos para renderizar de forma condicional un pequeño bloque de texto.

```
render() {
  const isLoggedIn = this.state.isLoggedIn;
  return (
    <div>
      The user is <b>{isLoggedIn ? 'currently' : 'not'}</b> logged in.
    </div>
  );
}
```

Cómo evitar la renderización de un componente

En casos excepcionales, es posible que necesitemos que un componente se oculte a sí mismo aunque haya sido renderizado por otro componente. Para hacer esto, debemos devolver null en lugar del resultado de renderizado.

```
function WarningBanner(props) {
  if (!props.warn) {
    return null;
  }
  return (
    <div className="warning">
      Warning!
    </div>
  );
}

class Page extends React.Component {
  constructor(props) {
```

```

    super(props);
    this.state = {showWarning: true};
    this.handleClick = this.handleClick.bind(this);
  }
  handleClick() {
    this.setState(state => ({
      showWarning: !state.showWarning
    }));
  }
  render() {
    return (
      <div>
        <WarningBanner warn={this.state.showWarning} />
        <button onClick={this.handleClick}>
          {this.state.showWarning ? 'Hide' : 'Show'}
        </button>
      </div>
    );
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Page />);

```

Iteraciones

En React, se puede iterar elementos o componentes usando **map** o javascript. En React, todo es javascript puro. Podemos usar métodos nativos de javascript que ejecuten una función. La asignación de componentes a variables es completamente válido también.

```

const mealsList = meals.map(meal => <MealItem key={meal.id} name={meal.name}
description={meal.description} />)
//Then use the mealsList in the component as:-

function AvailableMeals() {
  return (
    <ul>{mealsList}</ul>
  )
}

```

Renderizado de listas

Para renderizar listas se utilizan funcionalidades de javascript como los bucles *for* o la función *map()* tal como se define en siguiente ejemplo:

```
const products = [
  { title: 'Cabbage', id: 1 },
  { title: 'Garlic', id: 2 },
  { title: 'Apple', id: 3 },
];
```

Dentro del componente, utilizaremos la función `map()` para transformar el arreglo de productos en un arreglo de elementos ``:

```
const listItems = products.map(product =>
  <li key={product.id}>
    {product.title}
  </li>
);

return (
  <ul>{listItems}</ul>
);
```

Observemos que `` tiene un atributo `key` (identificador único). Para cada elemento en una lista, se debe pasar una cadena o un número que identifique ese elemento de forma única entre sus hermanos. Usualmente, una `key` debe provenir de nuestros datos, como un ID de una base de datos. React dependerá de nuestras llaves para entender qué ha ocurrido si luego insertamos, eliminamos o reordenamos los elementos.

Context

Context provee una forma de pasar datos a través del árbol de componentes sin tener que pasar props manualmente en cada nivel.

En una aplicación típica de React, los datos se pasan de arriba hacia abajo (de padre a hijo) a través de props, pero esta forma puede resultar incómoda para ciertos tipos de props (por ejemplo, localización, el tema de la interfaz) que son necesarias para muchos componentes dentro de una aplicación. Context proporciona una forma de compartir valores como estos entre componentes sin tener que pasar explícitamente una prop a través de cada nivel del árbol.

Cuándo usar Context

Context está diseñado para compartir datos que pueden considerarse "globales" para un árbol de componentes en React, como el usuario autenticado actual, el tema o el idioma preferido. Por ejemplo, en el código a continuación, pasamos manualmente una prop de "tema" para darle estilo al componente Button:

```
import React, { useContext } from 'react';

// Primero, creamos un Context.
const ThemeContext = React.createContext('light');
```

```
// Luego creamos un componente que utiliza el Context.
function ThemedButton() {
  const theme = useContext(ThemeContext);
  return <button theme={theme}>Soy un botón con tema {theme}</button>;
}

// Finalmente, creamos un componente que provee el Context.
function App() {
  return (
    <ThemeContext.Provider value="dark">
      <ThemedButton />
    </ThemeContext.Provider>
  );
}

export default App;
```

Usando Context podemos evitar pasar props a través de elementos intermedios:

Context se usa principalmente cuando algunos datos tienen que ser accesibles por muchos componentes en diferentes niveles de anidamiento. Se debe aplicar con moderación porque hace que la reutilización de componentes sea más difícil.

React.createContext()

```
const MyContext = React.createContext(defaultValue);
```

Crea un objeto Context. Cuando React renderiza un componente que se suscribe a este objeto Context, este leerá el valor de contexto actual del Provider más cercano en el árbol.

El argumento defaultValue es usado únicamente cuando un componente no tiene un Provider superior a él en el árbol. Este valor por defecto puede ser útil para probar componentes de forma aislada sin contenerlos.

Context.Provider

```
<MyContext.Provider value={/* algún valor */}>
```

Cada objeto Context viene con un componente Provider de React que permite que los componentes que lo consumen se suscriban a los cambios del contexto.

El componente Provider acepta una prop value que se pasará a los componentes consumidores que son descendientes de este Provider. Un Provider puede estar conectado a muchos consumidores. Los Providers pueden estar anidados para sobrescribir los valores más profundos dentro del árbol.

Todos los consumidores que son descendientes de un Provider se vuelven a renderizar cada vez que cambia la prop value del Provider. La propagación del Provider a sus consumidores descendientes (incluyendo

.contextType y useContext) no está sujeta al método shouldComponentUpdate, por lo que el consumidor se actualiza incluso cuando un componente padre evita la actualización.

Lifting State Up en React (Movimiento de estado hacia arriba en React)

Como hemos visto, cada componente en React tiene su propio estado, por este motivo a veces los datos pueden ser redundantes e inconsistentes. Entonces al notificar el estado de un componente hijo al componente padre, permitimos que el componente padre sea la única fuente de información segura y pasamos los datos del padre a sus hijos. Esto se usa frecuentemente para que los componentes estén sincronizados.

Por ejemplo, tenemos el siguiente componente


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

export default function MyApp() {
  return (
    <div>
      <h1>Los contadores de los botones se actualizan de manera independiente</h1>
      <MyButton />
      <MyButton />
    </div>
  );
}

function MyButton() {
  const [count, setCount] = useState(0);

  function handleClick() {
    setCount(count + 1);
  }

  return (
    <button onClick={handleClick}>
      Clicked {count} veces
    </button>
  );
}
```

y deseamos ahora que nuestra aplicación cuente con 2 componentes *MyButton*, que compartan datos y se actualicen siempre en conjunto. Para hacer que ambos componentes *MyButton* muestren el mismo count y se actualicen juntos, necesitas mover el estado de los botones individuales «hacia arriba» al componente más cercano que los contiene a todos.  Imagen de estado hacia arriba

En este ejemplo, *MyApp*s debería definir de la siguiente manera:

```
export default function MyApp() {
  const [count, setCount] = useState(0); //El estado de contador de click está
  ahora en el padre
```

```
function handleClick() { //El manejador de click de los botones, también está en
  el padre
  setCount(count + 1);
}

return (
  <div>
    <h1>Counters that update together</h1>
    <MyButton count={count} onClick={handleClick} />
    <MyButton count={count} onClick={handleClick} />
  </div>
);
}
```

Ahora cuando realizamos clic en cualquiera de los botones, *count* en *MyApp* cambiará, lo que causará que cambien ambos counts en *MyButton*. Esto se logró moviendo el estado de *MyButton* hacia *MyApp*.

Luego, se pasa el estado hacia abajo desde *MyApp* hacia cada *MyButton*, junto con la función compartida para manejar el evento de clic. Se puede pasar la información a *MyButton* usando las llaves de JSX. La información que se pasa hacia abajo se llaman *props*. Ahora el componente *MyApp* contiene el estado *count* y el manejador de eventos *handleClick*, y pasa ambos hacia abajo como *props* a cada uno de los botones.

Finalmente, el código de *myButton* para que pueda recibir las props del padre se define como:

```
function MyButton({ count, onClick }) {
  return (
    <button onClick={onClick}>
      Clicked {count} times
    </button>
  );
}
```

Cuando hacemos clic en el botón, el manejador *onClick* se dispara. A la prop *onClick* de cada botón se le asignó la función *handleClick* dentro de *MyApp*, de forma que el código dentro de ella se ejecuta. Ese código llama a *setCount(count + 1)*, que incrementa la variable de estado *count*. El nuevo valor de *count* se pasa como *prop* a cada botón, y así todos muestran el nuevo valor.

Esto se llama «levantar el estado hacia arriba». Al mover el estado hacia arriba, lo compartimos entre componentes.

Código completo aquí:

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

export default function MyApp() {
  const [count, setCount] = useState(0);

  function handleClick() {
```

```
    setCount(count + 1);
  }

  return (
    <div>
      <h1>Counters that update together</h1>
      <MyButton count={count} onClick={handleClick} />
      <MyButton count={count} onClick={handleClick} />
    </div>
  );
}

function MyButton({ count, onClick }) {
  return (
    <button onClick={onClick}>
      Clicked {count} times
    </button>
  );
}
```