APLICACIÓN DE HOOKS Y PROPS

useEffect – ABM usuarios

1

Explicación breve

Concepto

- useEffect es un **hook** de React que nos permite realizar **efectos secundarios** en nuestros componentes funcionales.
- Un efecto secundario es cualquier acción que **afecta al mundo externo de React**, como por ejemplo:
  - Llamar a una API.
  - Manipular directamente el DOM.
  - Establecer temporizadores o suscripciones.
  - Leer o escribir en localStorage.

- ¿Por qué usarlo?
  Porque en React ***cada vez que algo cambia (como un estado o una propiedad), el componente se vuelve a ejecutar.***
  Eso incluye ***todo el código que está dentro del cuerpo del componente.***.

2

Explicación breve

- Por ejemplo, si realizas una llamada a una API Externa desde un componente

```
function MiComponente() {
  fetch("https://api.com/datos") // ¡Esto se ejecutará cada vez que se renderice!
  return <div>Hola</div>;
}
```

- La API se EJECUTARÁ cada vez que el componente se vuelva a RENDERIZAR
- Con useEffect, puedes indicar por ejemplo:
  - *Que la API se ejecute LUEGO de que el componente SE RENDERICE*
  - *Que la API se ejecute cuando SEA NECESARIO.*

3

Sintaxis básica

```
useEffect(() => {
  // Aquí va el efecto
  return () => {
    // Esta es la función de limpieza (opcional)
  };
}, [dependencias]);
```

- Si el array de dependencias está vacío ([]), se ejecuta solo una vez al montar el componente.
- Si tiene valores, se ejecuta cada vez que esos valores cambien.
- Si no pones array, se ejecuta en **cada render**.

4

Ejemplo de bibliografía: Temporizador

Un temporizador básicamente es un contador (counter) que almacena un valor que se va actualizando de manera automática. Es un buen ejemplo para aplicar useEffect.

≠

Código	¿Quién lo ejecuta?	¿Cuándo se ejecuta?
`useEffect(..., [])`	React	Una vez, al montar
`setInterval(() => {...}, 1000)`	JavaScript del navegador	Cada segundo
`setCount(...)`	React	Cuando tú lo llamas

`prev` significa **el valor anterior** del estado `count`.

≠

`setCount(count + 1)`

src > components > counter.jsx > default

```jsx
import { useEffect, useState } from "react";

const Counter = ()=>{

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

    useEffect(()=>{

        const timer = setInterval(()=>{
            setCount(prev=>prev+1);
        },1000);

        return ()=>clearInterval(timer);
    },[]);

    return (
        <div className="counter-container">
            <h1>Contador</h1>
            <p className="counter-value">{count} segundos</p>
        </div>
    );

};

export default Counter;
```

5

.

Ejemplo: Control del ciclo de vida de un componente

- Vamos a leer los datos almacenados en un archivo de persistencia. En este caso un localStorage
- Un localStorage es una API del navegador que permite **almacenar datos clave-valor** localmente en el navegador del usuario. Es persistente: si recargas la página, los datos siguen ahí.
- En vez de una API real, usamos localStorage para:
  - **Inicializar una lista de usuarios de ejemplo.**
  - **Simular una fuente externa de datos.**
- Esto permite practicar la estructura de una app real sin necesidad de backend

6

.

Ejemplo: Control del ciclo de vida de un componente

src > services > JS userService.js > ...

```
1  const USERS_KEY = "userList";
2
3  export const getUsers = ()=>{
4      const data = localStorage.getItem(USERS_KEY);
5      return data ? JSON.parse(data) : [];
6  };
7
8  // Inicializa una lista de ejemplo si no existe
9  export const initializeUsers = () => {
10   const existing = localStorage.getItem(USERS_KEY);
11   if (!existing) {
12     const sampleUsers = [
13       { id: 1, name: 'Juan Pérez', email: 'juan@gmail.com' },
14       { id: 2, name: 'María Gómez', email: 'maria@gmail.com' },
15       { id: 3, name: 'Carlos Ruiz', email: 'carlos@gmail.com' }
16     ];
17     localStorage.setItem(USERS_KEY, JSON.stringify(sampleUsers));
18   }
19 };
```

```
return data ? JSON.parse(data) : [];
```

- Operador ternario equivalente a un if else, donde:
  - Si data **tiene algún valor** (es decir, no es null), **JSON.parse(data)**, transforma el string guardado en localStorage en un array real de objetos JavaScript.
  - Si data **es null**, devuelve un arreglo vacío para que el programa no tire error.

```
JSON.stringify(sampleUsers)
```

- Convierte el arreglo a un string JSON (un texto plano), que es lo único que localStorage puede guardar

7

.

Ejemplo: Control del ciclo de vida de un componente

src > components > UserList.jsx > ...

```
1  import { useEffect, useState } from "react";
2  import { getUsers, initializeUsers } from "../services/userService";
3  import "../components/UserList.css";
4
5  export const UserList = ()=>{
6      const [users, setUsers] = useState([]);
7
8      useEffect(()=>{
9          initializeUsers();
10         const data = getUsers();
11         setUsers(data);
12     },[]);
13
14     return(
15         <div className="userlist-container">
16             <h2>Lista de Usuarios</h2>
17             {
18                 users.length > 0 ? (
19                     <ul>
20                         {users.map(user=>(
21                         <li key={user.id} className="user-item">
22                             <strong>{user.name}</strong> - {user.email}
23                         </li>))}
24                     </ul>
25                 ):(
26                     <p>No hay usuarios registrados.</p>
27                 )
28             }
29         </div>
30     );
31 };
```

- El componente muestra la lista de usuarios que obtiene desde el localStorage.
- Para ello aplica el siguiente orden de ejecución;
  - Declara el estado
    `const [users, setUsers] = useState([]);`
  - Se renderiza por primera vez. Como el valor del estado de users es [] se debe visualizar
    `<p>No hay usuarios registrados.</p>`
  - Se ejecuta una única vez useEffect, porque el arreglo de dependencias es []:
    - Inicializa el localStorage de ser necesario
    - Obtiene los datos del localStorage como un arreglo de objetos
    - Actualiza el estado de users con ese arreglo de objetos
  - Como el estado se actualizó, se renderiza la lista de elementos

8

Otras dependencias: un valor

- Se ejecuta cuando el único valor indicado en el arreglo de dependencias cambia.
- Ejemplo: Mostrar por consola el valor de un contador cada vez que se actualice cuando el usuario presiona un botón.

🔍 Este useEffect **no se ejecuta al azar**: solo corre cuando cambia count. Es útil si quieres hacer una acción **en respuesta a un cambio** (ejemplo: guardar cambios, validar, animar, etc).

src > components > Counter.jsx > ...

```jsx
import { useEffect, useState } from 'react';
import '../components/styles/Counter.css';

export const Counter = () => {
  const [count, setCount] = useState(0);

  useEffect(() => {
    console.log(`El contador cambió a: ${count}`);
  }, [count]); // Se ejecuta cada vez que cambia "count"

  return (
    <div className="counter-container">
      <h2>Contador: {count}</h2>
      <button onClick={() => setCount(c => c + 1)}>Incrementar</button>
    </div>
  );
};
```

9

Otras dependencias: sin dependencias

- Se ejecuta en cada render.
- Esto es raro de usar, y solo tiene sentido para efectos globales como:
  - *medir FPS*
  - *Leer algo del DOM*
  - *debugging avanzado.*
- Ejemplo: medir cuánto tarda React en montar y renderizar visualmente un componente. Esto es algo que se podría usar para debugging o performance testing.

👄 **Precaución**: si escribes setState dentro de este efecto, vas a entrar en un **bucle infinito**.

src > components > RenderTimer.jsx > ...

```jsx
import { useEffect, useRef, useState } from 'react';
import '../components/styles/RenderTimer.css';

export const RenderTimer = () => {
  const startTime = useRef(performance.now());
  const [value, setValue] = useState("");

  useEffect(() => {
    const endTime = performance.now();
    const duration = endTime - startTime.current;
    console.log(`Renderizado completo en: ${duration.toFixed(2)} ms`);
  }); // Sin dependencias → corre en cada render

  return (
    <div className="render-timer-container">
      <h2>Medidor de Renderizado</h2>
      <input
        type="text"
        placeholder="Escribí algo"
        value={value}
        onChange={(e) => setValue(e.target.value)}
      />
      <p>Texto actual: {value}</p>
    </div>
  );
};
```

10

Otras dependencias: sin dependencias

- Muestra un <input>controlado por React.
- Cada vez que el usuario escribe algo el estado (value) cambia.
- Cada cambio provoca un nuevo render y React determina cuanto tardó en renderizar.

```
const startTime = useRef(performance.now());
```

- useRef: devuelve un objeto mutable persistente entre renders

src > components > RenderTimer.jsx > ...

```jsx
import { useEffect, useRef, useState } from 'react';
import '../components/styles/RenderTimer.css';

export const RenderTimer = () => {
  const startTime = useRef(performance.now());
  const [value, setValue] = useState("");

  useEffect(() => {
    const endTime = performance.now();
    const duration = endTime - startTime.current;
    console.log(`Renderizado completo en: ${duration.toFixed(2)} ms`);
  }); // Sin dependencias → corre en cada render

  return (
    <div className="render-timer-container">
      <h2>Medidor de Renderizado</h2>
      <input
        type="text"
        placeholder="Escribí algo"
        value={value}
        onChange={(e) => setValue(e.target.value)}
      />
      <p>Texto actual: {value}</p>
    </div>
  );
};
```

11

Resumen dependencias de useEffect

Dependencias	Cuándo se ejecuta
`[]`	Solo al **montar**
`[valor]`	Cuando **ese valor cambia**
`[a, b]`	Cuando **a o b cambian**
(sin dependencias)	En **cada renderizado**

12

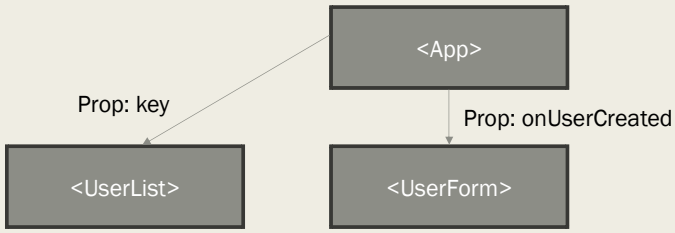
Ejemplo: ABM usuarios (Alta usuarios)

- Se almacenará en un localStorage
- Se creará un formulario funcional para agregar usuarios
- Se actualizará el estado de los componentes que mostrarán los nuevos usuarios agregados
- Se seguirán aplicando la separación en capas.

13

Ejemplo: ABM usuarios (Alta usuarios)

- Esta es la jerarquía de componentes dentro del ejemplo

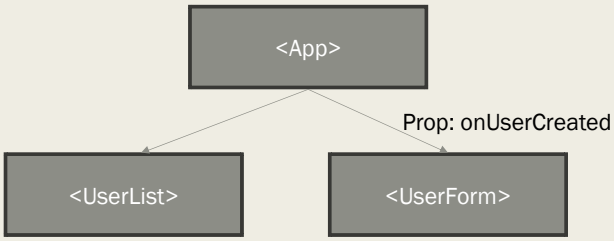


Un "prop" (abreviatura de "property" o propiedad) es una forma de pasar datos de un componente padre a un componente hijo. Son esencialmente argumentos que se pasan a los componentes cuando se crean y permiten la comunicación de datos entre ellos

14

Ejemplo: ABM usuarios (Alta usuarios)

- Puedes pasar una función como prop al componente hijo y luego, dentro del componente hijo, puedes llamar a esa función para que se ejecute.

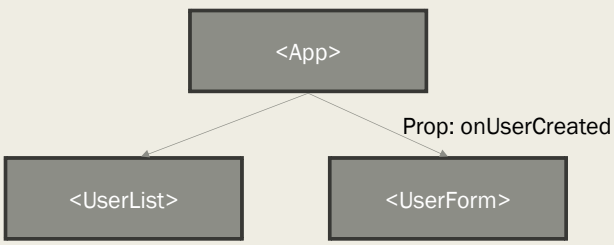



```jsx
function App() {

  const [reload, setReload] = useState(false);

  const handleUserCreated =()=>{
    setReload(!reload); //simple código para recargar lista
  };


  return (
    <>
      <Counter></Counter>
      <RenderTimer></RenderTimer>

      <h1>Gestión de Usuarios (ABM)</h1>
      <UserForm onUserCreated={handleUserCreated}/>
      <UserList key={reload}/>
    </>
  );
}

export default App
```

15

Ejemplo: Alta usuarios

- Puedes pasar una función como prop al componente hijo y luego, dentro del componente hijo, puedes llamar a esa función para que se ejecute.

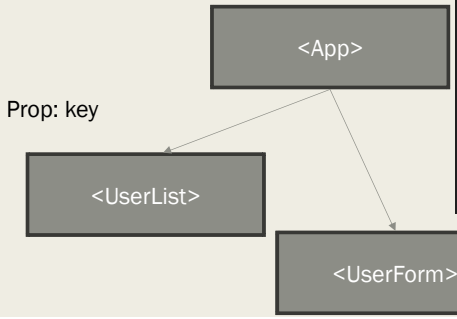



```jsx
export const UserForm = ({ onUserCreated }) => {
  const [name, setName] = useState("");
  const [email, setEmail] = useState("");

  const handleSubmit = (e) => {
    e.preventDefault();
    const newUser = {
      id: Date.now(), // identificador único
      name,
      email
    };
    saveUser(newUser);
    onUserCreated(); // notifica al padre
    setName("");
    setEmail("");
  };

  return (
    <form className="user-form" onSubmit={handleSubmit}>
      <h3>Agregar Usuario</h3>
      <input
        type="text"
        placeholder="Nombre"
        value={name}
        onChange={(e) => setName(e.target.value)}
        required
      />
      <input
        type="email"
        placeholder="Correo"
        value={email}
        onChange={(e) => setEmail(e.target.value)}
        required
      />
      <button type="submit">Guardar</button>
    </form>
  );
```

16

Ejemplo: Alta usuarios

- React usa la prop key para identificar componentes. Si key cambia, React desmonta y vuelve a montar el componente, lo que fuerza a que se ejecute de nuevo su useEffect. Es una estrategia rápida (aunque no la más fina) para recargar el componente.



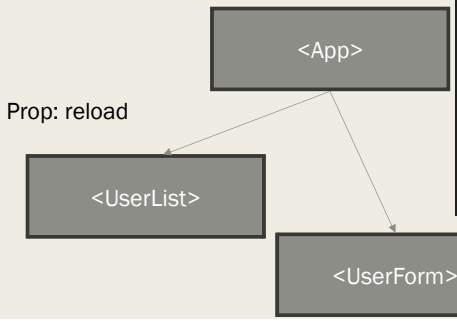

```jsx
function App() {

  const [reload, setReload] = useState(false);

  const handleUserCreated =()=>{
    setReload(!reload); //simple código para recargar lista
  };


  return (
    <>
      <Counter></Counter>
      <RenderTimer></RenderTimer>

      <h1>Gestión de Usuarios (ABM)</h1>
      <UserForm onUserCreated={handleUserCreated}/>
      <UserList key={reload}/>
    </>
  );
}

export default App
```

```jsx
export const UserList = () => {
  const [users, setUsers] = useState([]);

  const fetchUsers = () => {
    const data = getUsers();
    setUsers(data);
  };

  useEffect(() => {
    fetchUsers();
  }, []);

  return (
    <div className="userlist-container">
      <h2>Usuarios Registrados</h2>
      {users.length > 0 ? (
        <ul>
          {users.map(user => (
            <li key={user.id} className="user-item">
              <strong>{user.name}</strong> - {user.email}
            </li>
          ))}
        </ul>
      ) : (
        <p>No hay usuarios todavía.</p>
      )}
    </div>
  );
};

export default UserList;
```

17

Ejemplo: Alta usuarios

- Lo más ordenado sería usar el useEffect con un valor para que se recargue el componente cada vez que ese valor cambia, en lugar de usar el artilugio de cambio de valor de la prop key del componente..




```jsx
function App() {

  const [reload, setReload] = useState(false);

  const handleUserCreated =()=>{
    setReload(!reload); //simple código para recargar lista
  };


  return (
    <>
      <Counter></Counter>
      <RenderTimer></RenderTimer>

      <h1>Gestión de Usuarios (ABM)</h1>
      <UserForm onUserCreated={handleUserCreated}/>
      <UserList key={reload}/>
    </>
  );
}

export default App
```

```jsx
export const UserList = ({ reload }) => {
```

```jsx
export const UserList = () => {
  const [users, setUsers] = useState([]);

  const fetchUsers = () => {
    const data = getUsers();
    setUsers(data);
  };

  useEffect(() => {
    fetchUsers();
  }, []);

  return (
    <div className="userlist-container">
      <h2>Usuarios Registrados</h2>
      {users.length > 0 ? (
        <ul>
          {users.map(user => (
            <li key={user.id} className="user-item">
              <strong>{user.name}</strong> - {user.email}
            </li>
          ))}
        </ul>
      ) : (
        <p>No hay usuarios todavía.</p>
      )}
    </div>
  );
};

export default UserList;
```

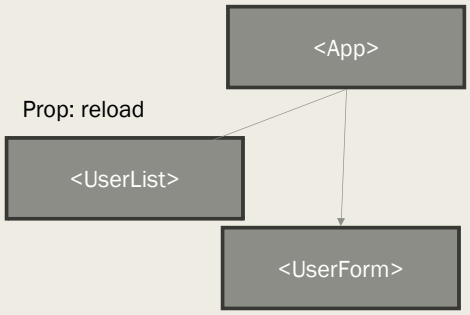
18

.

Ejemplo: Alta usuarios

✅ **Ventajas del enfoque con reload como prop:**

- Más controlado y explícito: React **no desmonta** el componente como hace con key.
- Es más **legible** para quienes comienzan: se ve claramente que UserList depende del valor de reload.
- Es **fácil de extender** si más adelante quieres cargar desde API, aplicar filtros, etc.




```
function App() {

  const [reload, setReload] = useState(false);

  const handleUserCreated =()=>{
    setReload(!reload); //simple código para recargar lista
  };


  return (
    <>
      <Counter></Counter>
      <RenderTimer></RenderTimer>

      <h1>Gestión de Usuarios (ABM)</h1>
      <UserForm onUserCreated={handleUserCreated}/>
      <UserList key={reload}/>
    </>
  );
}

export default App
```

<UserList reload={reload} />

```
export const UserList = ({ reload }) => {
```

```
export const UserList = () => {
  const [users, setUsers] = useState([]);

  const fetchUsers = () => {
    const data = getUsers();
    setUsers(data);
  };

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

}, [reload]);

```
  return (
    <div className="userlist-container">
      <h2>Usuarios Registrados</h2>
      {users.length > 0 ? (
        <ul>
          {users.map(user => (
            <li key={user.id} className="user-item">
              <strong>{user.name}</strong> - {user.email}
            </li>
          ))}
        </ul>
      ) : (
        <p>No hay usuarios todavía.</p>
      )}
    </div>
  );
};

export default UserList;
```

19