React Nivel Avanzado: Manejo del Estado con useReducer 📝

En este nivel, aprenderás a utilizar el hook `useReducer` para manejar estados complejos en tus componentes de React. `useReducer` es una alternativa a `useState` que resulta más adecuada cuando tienes lógica de estado más compleja o necesitas agrupar varias actualizaciones de estado. Veremos ejemplos prácticos y aprenderemos cuándo usarlo para simplificar la gestión del estado en aplicaciones React más grandes.

Temas que Cubriremos 📚

1. ¿Qué es `useReducer` y cuándo usarlo?
2. Comparación entre `useState` y `useReducer`.
3. Creación y uso de un reducer.
4. Dispatch de acciones: cómo actualizar el estado.
5. Ejemplo práctico de una lista de tareas con `useReducer`.

---

1. ¿Qué es `useReducer` y Cuándo Usarlo? 🤔

`useReducer` es un hook que te permite manejar el estado de un componente usando un patrón de reducer. Es ideal cuando:

- Tienes estados complejos que involucran múltiples variables.
- Necesitas aplicar lógica condicional para actualizar el estado.
- Quieres organizar mejor la lógica de actualización del estado en un solo lugar.

La estructura básica es:

```
const [state, dispatch] = useReducer(reducer, estadoInicial);
```

- `state`: El estado actual gestionado por el reducer.
- `dispatch`: Función para enviar acciones al reducer.
- `reducer`: Función que define cómo actualizar el estado.
- `estadoInicial`: El valor inicial del estado.

---

2. Comparación entre `useState` y `useReducer` ⚖️

`useState`	`useReducer`
Fácil de usar para estados simples.	Ideal para lógica de estado compleja.
Actualización directa del estado.	Usa una función reducer para actualizar el estado.
Menos código cuando hay pocas variables.	Código más estructurado y fácil de mantener.

Cuándo usar `useReducer`:

- Cuando tienes múltiples valores de estado relacionados.
- Cuando la lógica de actualización del estado es compleja.
- Para manejar formularios o listas con múltiples acciones (agregar, editar, eliminar).

---

3. Creación y Uso de un Reducer 🛠️

Un reducer es una función que recibe el estado actual y una acción, y devuelve un nuevo estado basado en la acción.

Ejemplo:

```
const reducer = (state, action) => {
  switch (action.type) {
    case "INCREMENTAR":
      return { count: state.count + 1 };
    case "DECREMENTAR":
      return { count: state.count - 1 };
    default:
      return state;
  }
};

const estadoInicial = { count: 0 };
```

En este ejemplo, `reducer` maneja dos acciones: `INCREMENTAR` y `DECREMENTAR`, que modifican el valor de `count`.

---

4. Dispatch de Acciones: Cómo Actualizar el Estado 🔄

El `dispatch` se usa para enviar acciones al reducer. Cada acción es un objeto con un `type` que indica qué acción realizar y opcionalmente un `payload` con datos adicionales.

Ejemplo:

```
import React, { useReducer } from "react";

function Contador() {
  const estadoInicial = { count: 0 };

  const reducer = (state, action) => {
    switch (action.type) {
      case "INCREMENTAR":
        return { count: state.count + 1 };
      case "DECREMENTAR":
        return { count: state.count - 1 };
      default:
        return state;
    }
  };
```

```
  const [state, dispatch] = useReducer(reducer, estadoInicial);

  return (
    <div>
      <h1>Contador: {state.count}</h1>
      <button onClick={() => dispatch({ type: "INCREMENTAR" })}>Incrementar</button>
      <button onClick={() => dispatch({ type: "DECREMENTAR" })}>Decrementar</button>
    </div>
  );
}

export default Contador;
```

Explicación:

- `dispatch({ type: "INCREMENTAR" })` envía la acción al reducer, que actualiza el estado sumando 1.
- El componente se vuelve a renderizar con el nuevo valor del estado.

---

5. Ejemplo Práctico: Lista de Tareas con `useReducer` ✅

En este ejemplo, vamos a crear una lista de tareas donde podrás agregar, editar y eliminar tareas usando `useReducer`.

```
import React, { useReducer } from "react";

const reducer = (state, action) => {
  switch (action.type) {
    case "AGREGAR_TAREA":
      return [...state, action.payload];
    case "ELIMINAR_TAREA":
      return state.filter((_, index) => index !== action.payload);
    case "EDITAR_TAREA":
      const nuevasTareas = [...state];
      nuevasTareas[action.payload.index] = action.payload.texto;
      return nuevasTareas;
    default:
      return state;
  }
};

const estadoInicial = [];

function ListaDeTareas() {
  const [tareas, dispatch] = useReducer(reducer, estadoInicial);
  const [nuevaTarea, setNuevaTarea] = React.useState("");
  const [textoEditado, setTextoEditado] = React.useState("");
  const [editandoIndex, setEditandoIndex] = React.useState(null);

  const agregarTarea = () => {
    if (nuevaTarea.trim()) {
      dispatch({ type: "AGREGAR_TAREA", payload: nuevaTarea });
```

```
      setNuevaTarea("");
    }
  };

  const eliminarTarea = (index) => dispatch({ type: "ELIMINAR_TAREA", payload: index
});

  const editarTarea = (index) => {
    setEditandoIndex(index);
    setTextoEditado(tareas[index]);
  };

  const guardarEdicion = (index) => {
    if (textoEditado.trim()) {
      dispatch({ type: "EDITAR_TAREA", payload: { index, texto: textoEditado } });
      setEditandoIndex(null);
      setTextoEditado("");
    }
  };

  return (
    <div>
      <h1>Lista de Tareas</h1>
      <input
        type="text"
        value={nuevaTarea}
        onChange={(e) => setNuevaTarea(e.target.value)}
        placeholder="Nueva Tarea"
      />
      <button onClick={agregarTarea}>Agregar</button>
      <ul>
        {tareas.map((tarea, index) => (
          <li key={index}>
            {editandoIndex === index ? (
              <>
                <input
                  type="text"
                  value={textoEditado}
                  onChange={(e) => setTextoEditado(e.target.value)}
                />
                <button onClick={() => guardarEdicion(index)}>Guardar</button>
              </>
            ) : (
              <>
                {tarea}{" "}
                <button onClick={() => editarTarea(index)}>Editar</button>
                <button onClick={() => eliminarTarea(index)}>Eliminar</button>
              </>
            )}
          </li>
        ))}
      </ul>
    </div>
  );
}
```

```
export default ListaDeTareas;
```

Conclusión:

- `useReducer` centraliza la lógica del estado en una sola función (`reducer`), haciendo el código más fácil de mantener.
- Permite manejar estados complejos con múltiples acciones.
- Es especialmente útil para componentes que requieren lógica de estado avanzada, como formularios o listas de tareas.

Resumen 🏁

En este nivel, aprendiste a manejar el estado de forma más avanzada usando `useReducer`. Ahora estás listo para manejar estados complejos y estructurados en tus aplicaciones React. ¡Sigue practicando y experimentando para dominar estos conceptos! 🚀💻