React: Hooks y Manejo de Estado

1. Introducción a los Hooks

Los Hooks fueron introducidos en React 16.8 y permiten usar estado y otras características de React sin escribir clases.

Reglas de los Hooks:
1. Solo llamar Hooks en el nivel superior (no dentro de loops, condiciones o funciones anidadas)
2. Solo llamar Hooks desde componentes funcionales de React o desde otros Hooks personalizados

Los Hooks principales son:
- useState: para manejar estado local
- useEffect: para efectos secundarios
- useContext: para consumir contexto
- useReducer: para estado complejo
- useCallback: para memorizar funciones
- useMemo: para memorizar valores computados
- useRef: para referencias mutables

Ejemplo básico de componente con Hooks:
```
function Contador() {
  const [count, setCount] = useState(0);

  return (
    <button onClick={() => setCount(count + 1)}>
      Clicks: {count}
    </button>
  );
}
```

2. useState en Profundidad

useState es el Hook más básico para manejar estado en componentes funcionales.

Sintaxis:
```
const [estado, setEstado] = useState(valorInicial);
```

El valor inicial puede ser cualquier tipo de dato:
```
const [nombre, setNombre] = useState("");
const [edad, setEdad] = useState(0);
const [activo, setActivo] = useState(true);
const [usuario, setUsuario] = useState({ nombre: "", email: "" });
const [items, setItems] = useState([]);
```

Actualización funcional (cuando el nuevo valor depende del anterior):
```
setCount(prevCount => prevCount + 1);
```

Esto es importante cuando se hacen múltiples actualizaciones:

```
// Incorrecto - solo incrementa 1
setCount(count + 1);
setCount(count + 1);

// Correcto - incrementa 2
setCount(prev => prev + 1);
setCount(prev => prev + 1);
```

Para objetos, siempre crear nuevo objeto (inmutabilidad):

```
setUsuario({ ...usuario, nombre: "Nuevo nombre" });
```

Para arrays:

```
// Añadir
setItems([...items, nuevoItem]);
// Eliminar
setItems(items.filter(item => item.id !== idAEliminar));
// Actualizar
setItems(items.map(item =>
  item.id === id ? { ...item, completado: true } : item
));
```

Inicialización perezosa (lazy initialization):

```
const [datos, setDatos] = useState(() => {
  // Solo se ejecuta en el primer render
  return calcularValorInicial();
});
```

3. useEffect para Efectos Secundarios

useEffect permite ejecutar efectos secundarios en componentes funcionales: llamadas API, suscripciones, manipulación del DOM, etc.

Sintaxis:

```
useEffect(() => {
  // Código del efecto
  return () => {
    // Cleanup (opcional)
  };
}, [dependencias]);
```

Casos de uso según dependencias:

1. Sin array de dependencias - se ejecuta en cada render:

```
useEffect(() => {
  console.log("Cada render");
});
```

2. Array vacío - solo en el montaje:

```
useEffect(() => {
  console.log("Solo al montar");
  return () => console.log("Al desmontar");
}, []);
```

3. Con dependencias - cuando cambian:

```
useEffect(() => {
  console.log("userId cambió:", userId);
  fetchUserData(userId);
}, [userId]);
```

Ejemplo práctico - fetch de datos:

```
function UserProfile({ userId }) {
  const [user, setUser] = useState(null);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    let cancelled = false;

    async function fetchUser() {
      setLoading(true);
      try {
        const response = await fetch("/api/users/" + userId);
        const data = await response.json();
        if (!cancelled) {
          setUser(data);
        }
      } finally {
        if (!cancelled) {
          setLoading(false);
        }
      }
    }

    fetchUser();

    return () => {
      cancelled = true; // Evita actualizaciones después de desmontar
    };
  }, [userId]);

  if (loading) return <div>Cargando...</div>;
  return <div>{user.name}</div>;
}
```

4. useContext para Estado Global

useContext permite consumir contexto sin necesidad de render props o Consumer.

Crear el contexto:

```
const ThemeContext = React.createContext("light");
```

Proveer el contexto:

```
function App() {
  const [theme, setTheme] = useState("light");

  return (
```

```
    <ThemeContext.Provider value={{ theme, setTheme }}>
      <MainContent />
    </ThemeContext.Provider>
  );
}

Consumir el contexto con useContext:
function Button() {
  const { theme, setTheme } = useContext(ThemeContext);

  return (
    <button
      style={{ background: theme === "dark" ? "#333" : "#fff" }}
      onClick={() => setTheme(theme === "dark" ? "light" : "dark")}
    >
      Toggle Theme
    </button>
  );
}

Patrón recomendado - crear un hook personalizado:
// ThemeContext.js
const ThemeContext = createContext();

export function ThemeProvider({ children }) {
  const [theme, setTheme] = useState("light");
  const toggleTheme = () => setTheme(t => t === "light" ? "dark" : "light");

  return (
    <ThemeContext.Provider value={{ theme, toggleTheme }}>
      {children}
    </ThemeContext.Provider>
  );
}

export function useTheme() {
  const context = useContext(ThemeContext);
  if (!context) {
    throw new Error("useTheme debe usarse dentro de ThemeProvider");
  }
  return context;
}

// Uso en componentes
function MiComponente() {
  const { theme, toggleTheme } = useTheme();
  // ...
}
```

5. useReducer para Estado Complejo

useReducer es preferible a useState cuando la lógica de estado es compleja o cuando el siguiente estado depende del anterior.

Sintaxis:
```
const [state, dispatch] = useReducer(reducer, initialState);
```

Ejemplo - carrito de compras:
```
const initialState = {
  items: [],
  total: 0
};

function cartReducer(state, action) {
  switch (action.type) {
    case "ADD_ITEM":
      return {
        ...state,
        items: [...state.items, action.payload],
        total: state.total + action.payload.price
      };
    case "REMOVE_ITEM":
      const item = state.items.find(i => i.id === action.payload);
      return {
        ...state,
        items: state.items.filter(i => i.id !== action.payload),
        total: state.total - (item?.price || 0)
      };
    case "CLEAR_CART":
      return initialState;
    default:
      return state;
  }
}

function ShoppingCart() {
  const [state, dispatch] = useReducer(cartReducer, initialState);

  const addItem = (item) => {
    dispatch({ type: "ADD_ITEM", payload: item });
  };

  const removeItem = (id) => {
    dispatch({ type: "REMOVE_ITEM", payload: id });
  };

  return (
    <div>
      <p>Total: {state.total}</p>
      {state.items.map(item => (
        <div key={item.id}>
          {item.name} - {item.price}
          <button onClick={() => removeItem(item.id)}>Eliminar</button>
        </div>
      ))}
    </div>
  );
```

```
}
```

useReducer es especialmente útil cuando se combina con Context para crear un store global similar a Redux pero sin dependencias externas.

6. Hooks de Optimización: useMemo y useCallback

Estos hooks ayudan a optimizar el rendimiento evitando cálculos o recreaciones innecesarias.

useMemo - memoriza valores computados:

```
const valorMemoizado = useMemo(() => {
  return calcularValorCostoso(a, b);
}, [a, b]); // Solo recalcula si a o b cambian
```

Ejemplo práctico:

```
function ListaFiltrada({ items, filtro }) {
  const itemsFiltrados = useMemo(() => {
    console.log("Filtrando..."); // Solo cuando items o filtro cambian
    return items.filter(item =>
      item.nombre.toLowerCase().includes(filtro.toLowerCase())
    );
  }, [items, filtro]);

  return (
    <ul>
      {itemsFiltrados.map(item => <li key={item.id}>{item.nombre}</li>)}
    </ul>
  );
}
```

useCallback - memoriza funciones:

```
const funcionMemoizada = useCallback(() => {
  hacerAlgo(a, b);
}, [a, b]);
```

Es útil cuando pasas callbacks a componentes hijos optimizados:

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

  // Sin useCallback, se crea nueva función en cada render
  const handleClick = useCallback(() => {
    console.log("Click!");
  }, []); // Función estable

  return (
    <>
      <p>{count}</p>
      <button onClick={() => setCount(c => c + 1)}>+1</button>
      <HijoMemoizado onClick={handleClick} />
    </>
  );
}
```

```
const HijoMemoizado = React.memo(function Hijo({ onClick }) {
  console.log("Hijo renderizado");
  return <button onClick={onClick}>Click me</button>;
});
```

Importante: No usar estos hooks prematuramente. Solo optimizar cuando hay problemas de rendimiento medibles. La memorización tiene su propio costo.